The versatility of using explanations within constraint programming

Narendra Jussien

The versatility of using explanations within constraint programming

Narendra JUSSIEN jussien@emn.fr
École des Mines de Nantes, 4 rue Alfred Kastler, BP 20722,
F-44307 Nantes Cedex 3, France

Résumé

La programmation par contraintes est un sujet de recherche qui tire profit de nombreuses autres disciplines : mathématiques discrètes, analyse numérique, intelligence artificielle, recherche opérationnelle et calcul formel. Elle a prouvé son intérêt et son efficacité dans de nombreux domaines : optimisation combinatoire, ordonnancement, finance, simulation et synthèse de composants, diagnostic de panne, biologie moléculaire, ou encore problèmes géométriques. Néanmoins, un certain nombre de limitations et de difficultés ont été identifiées dans le domaine : conception d'algorithmes génériques et stables, traitement des problèmes dynamiques, accessibilité des concepts et des outils, ...

Dans ce document, nous plaidons pour l'utilisation de la notion d'explication au sein de la programmation par contraintes. Notre but est double : non seulement présenter un tableau général des explications (définition, génération et utilisations) mais aussi montrer comment leur utilisation permet de contribuer à lever certains des problèmes ouverts en programmation par contraintes. Nous présentons aussi une démarche générale de résolution de problème dans un environnement expliqué. Enfin, nous montrons comment ce nouveau sujet semble promis à un bel avenir.

Abstract

Constraint programming is a research topic benefiting from many other areas: discrete mathematics, numerical analysis, artificial intelligence, operations research, and formal calculus. It has proven its interest and its efficiency in various domains: combinatorial optimization, scheduling, finance, simulation and synthesis, diagnosis, molecular biology, or geometrical problems. However, some limitations and difficulties remain: designing stable and generic algorithms, handling dynamic problems, opening constraint programming to non-specialists, etc.

In this document, we advocate the use of explanations within constraint programming. Our aim is two-fold: drawing the *big picture* about explanations (definition, generation, management and use) and showing that they can help address several issues in constraint programming. We also introduce a new general explanation-based search technique that has been successfully used to design new efficient algorithms. Finally, current open issues and research topics in this field are presented.

Contents

I. Introduction

Constraint programming is a research topic that benefits from various other areas: discrete mathematics, numerical analysis, artificial intelligence, operations research, and formal calculus. It has proven its interest and its efficiency in various domains: combinatorial optimization, scheduling, finance, simulation and synthesis, diagnosis, molecular biology, or geometrical problems.

The efficiency of constraint programming is due to several reasons:

- one of the hypotheses of constraint programming, search of solutions in a bounded domain, is often naturally verified in practice where unknown values represent physical values;
- tools and methods are of very general application, they are not restricted to a particular problem;
- developed techniques remain usable even when considering modifications of the structure of the used constraints evolves.

However, there are still limitations to constraint programming. Many difficulties have not yet been satisfactorily solved:

- designing stable and generic techniques for solving any kind of problem;
- handling dynamic problems (for which the constraint system evolves): how not to solve a problem from scratch each time a modification occurs, how to only undo what is necessary when handling modifications, etc;
- providing debugging and analysis tools for promoting constraint programming to non-specialists.

Using conflict sets (contradictory subsets of constraints) or explanations (subsets of constraints justifying solver events) is not a new idea in constraint programming. They have been used to design innovative search techniques: *Conflict-based BackJumping* (`cbj`) [66], *Dynamic Backtracking* [37], *Nogood recording* [72] and more recently `mac-dbt` [51] and `decision-repair` [52]. They (or related concepts) are used to design incremental algorithms for solving dynamic *Constraint Satisfaction Problems*: [11] and [50]. Finally, but surprisingly only recently, explanations have been used to introduce user interaction within constraint programming: either for debugging [55] or analysis purposes [75].

This versatility in the use of explanations within constraint programming advocates for their generalization in constraint solvers. The aim of this document is two-fold: to provide an overview of how explanations can be generated, managed and used in a constraint programming language and, secondly, as a result, to introduce their different possible usage and to show how they can overcome some limitations of constraint programming. In this document, we try to point out the main issues as well as to illustrate the main applications with our past experience. Moreover, we exhibit several important *side effects* of our work: from theoretical results (*e.g.* correctness of constraint retraction [26]) to practical one (*e.g.* a new family of search algorithms [52]).

This document is organized as follows:

- in the **first part**, the formal framework in which we are working is introduced in Section 1, then the notion of *explanation* is introduced in Section 2;
- in the **second part**, the way of using the theory of explanations to implement an explanation-based solver is presented in Section 3. Implementation issues are then addressed (Section 4), before introducing other existing approaches in Section 5;
- in the **third part**, the variety of possible uses of the notion of explanation is described from three angles: user information (Section 6), incremental handling of dynamic problems (Section 7) and search improvements (Section 8). Ending this final part, a case study of the open-shop scheduling problem (Section 9) is presented.

Notice that, in the following, we will use the choco[1] free constraint solver [59] as the basis for our implementations.

1. choco is an open source constraint engine developed as the kernel of the OCRE project. The OCRE project (its name stands for *Outil Contraintes pour la Recherche et l'Enseignement*, *i.e. constraint tool for research and education*) aims at building free Constraint Programming tools that anyone in the Constraint Programming community can use. For more information see choco-constraints.net.

II. Definitions

In this second part, we introduce the formal framework for constraint programming in which we define our explanations. Moreover, some related concepts are addressed.

1. Constraint Satisfaction Problems

Constraint satisfaction problems (CSP) [78] have proven to be an efficient model for solving many combinatorial and complex problems. We introduce here a formal model for representing both the constraint network and its resolution (domain reductions, constraint propagation and enumeration).

1.1 The constraint network

Following [78], a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function var : $C \to \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V. For the semantic part, we need to consider various *families* $f = (f_i)_{i \in I}$. Such a family is referred to by the *function* $i \mapsto f_i$ or by the *set* $\{(i, f_i) \mid i \in I\}$.

1.1.1 Domains

$(D_x)_{x \in V}$ is a family where each D_x is a *finite non-empty set* of possible values for x. We define the *domain of computation* by $\mathbb{D} = \bigcup_{x \in V}(\{x\} \times D_x)$. This domain allows simple and uniform definitions of (local consistency) operators on a power-set. For reduction, we consider subsets d of $\mathbb{D}$. Such a subset is called an *environment*. Let $d \subseteq \mathbb{D}$, $W \subseteq V$, we denote by $d|_W = \{(x, e) \in d \mid x \in W\}$. d is actually a family $(d_x)_{x \in V}$ with $d_x \subseteq D_x$: for $x \in V$, we define $d_x = \{e \in D_x \mid (x, e) \in d\}$. d_x is the *domain* of variable x.

1.1.2 Constraints as a set of allowed tuples

Constraints are defined by their set of allowed tuples. A *tuple* t on $W \subseteq V$ is a particular environment such that each variable of W appears only once: $t \subseteq \mathbb{D}|_W$ and $\forall x \in W, \exists e \in D_x, t|_{\{x\}} = \{(x, e)\}$. For each $c \in C$, T_c is a set of tuples on $\text{var}(c)$, called the solutions of c. Note that a tuple $t \in T_c$ is equivalent to a family $(e_x)_{x \in \text{var}(c)}$ and t is identified with $\{(x, e_x) \mid x \in \text{var}(c)\}$.

We can now formally define a CSP and a solution to it.

Definition 1 *A* Constraint Satisfaction Problem *(CSP) is defined by:*

- *a finite set V of variables;*
- *a finite set C of constraints;*
- *a function $var : C \to \mathcal{P}(V)$;*
- *a family $(D_x)_{x \in V}$ (the domains);*
- *a family $(T_c)_{c \in C}$ (the constraint semantics).*

Definition 2 *A* solution *for a* CSP $(V, C, var, (D_x)_{x \in V}, (T_c)_{c \in C})$ *is a tuple s on V such that $\forall c \in C, s|_{var(c)} \in T_c$.*

1.2 Domain reduction and propagation

Two more key concepts need some details: the domain reduction mechanism and the propagation mechanism itself.

1.2.1 Local propagators

A constraint is fully characterized by its behavior regarding modifications of the environments of the variables. *Local consistency operators* are associated with constraints. Such an operator has a *type* (W_{in}, W_{out}) with $W_{in}, W_{out} \subseteq V$. For the sake of clarity, we will consider in our formal presentation that each operator is applied to the whole environment but, in practice, it only removes from the environments of W_{out} some values which are inconsistent with respect to the environments of W_{in}.

Definition 3 *A* local consistency operator *of type* (W_{in}, W_{out})*, with* $W_{in}, W_{out} \subseteq V$*, is a monotonic function* $r : \mathcal{P}(\mathbb{D}) \rightarrow \mathcal{P}(\mathbb{D})$ *such that:* $\forall d \subseteq \mathbb{D}$, $r(d)|_{V \setminus W_{out}} = \mathbb{D}|_{V \setminus W_{out}}$, *and* $r(d) = r(d|_{W_{in}})$

Example 1 (Constraint $x \geq y + c$):
$x \geq y + c$ is one of the basic constraints in CHOCO. It is represented by the `GreaterOrEqualxyc` class. Reacting to an upper bound update for this constraint can be stated as: if the upper bound of x is modified then the upper bound of y should be lowered to the new value of the upper bound of x (taking into account the constant c). This is encoded as:

```
[awakeOnSup(c:GreaterOrEqualxyc,idx:integer)
 -> if (idx = 1)
       updateSup(c.v2,c.v1.sup - c.cste)]
```

`idx` is the index of the variable of the constraint whose bound (the upper bound here) has been modified. This particular constraint only reacts to modification of the upper bound of variable x (`c.v1` in the CHOCO representation of the constraint). The `updateSup` method only modifies the value of y (`c.v2` in the constraint) when the upper bound is really updated.
The `awakeOnSup` method can be considered as a local consistency operator with $W_{in} = \{\texttt{c.v1}\}$ and $W_{out} = \{\texttt{c.v2}\}$.

Classically [79, 31, 9, 6], reduction operators are considered as *monotonic*, *contractant* and *idempotent* functions. However, on the one hand, *contractance* is not mandatory because environment reduction after applying a given operator r can be forced by intersecting its result with the current environment, that is $d \cap r(d)$. On the other hand, *idempotence* is useless from a theoretical point of view (it is only useful in practice for managing the propagation queue). This is generally not mandatory to design effective constraint solvers. We can therefore use only *monotonic* functions in Definition 3.

The solver semantics are completely described by the set of such operators associated with the handled constraints. More or less accurate local consistency operators may be selected for each constraint. Moreover, this framework is not limited to arc-consistency but may handle any local consistency which boils down to domain reduction as shown in [32].

Of course local consistency operators should be *correct* with respect to the constraints. In practice, to each constraint $c \in C$ is associated a set of local consistency operators $R(c)$. The set $R(c)$ is such that for each $r \in R(c)$: let (W_{in}, W_{out}) be the type of r with $W_{in}, W_{out} \subseteq \text{var}(c)$; for each $d \subseteq \mathbb{D}$, $t \in T_c$: $t \subseteq d \Rightarrow t \subseteq r(d)$.

1.2.2 Constraint propagation

Propagation is handled through a propagation queue (containing events or conversely operators to awake). Informally, starting from the given *initial environment* for the problem, a local consistency operator is selected from the propagation queue (initialized with all the operators) and applied to

the environment resulting in a new one. If an environment/domain reduction occurs, new operators (or new events) are added to the propagation queue.

Termination is reached when:

1. a variable environment is emptied: there is no solution to the associated problem;
2. the propagation queue is emptied: a common fix-point (or a desired consistency state) is reached ensuring that further propagation will not modify the result.

The resulting environment is actually obtained by sequentially applying a given sequence of operators. To formalize this result, let us consider iterations.

Definition 4 *The* iteration *[6] from the initial environment $d \subseteq \mathbb{D}$ with respect to an* infinite *sequence of operators of R: $r^1, r^2, \ldots$ is the infinite sequence of environments $d^0, d^1, d^2, \ldots$ inductively defined by:*

- $d^0 = d$;
- $\forall i \in \mathbb{N}^*,\ d^{i+1} = d^i \cap r^{i+1}(d^i)$.

Its limit *is $\cap_{i \in \mathbb{N}} d^i$.*

A chaotic iteration *is an iteration with respect to a sequence of operators of R where each $r \in R$ appears infinitely often.*

The most accurate set that can be computed using a set of local consistency operators in the framework of domain reduction is the *downward closure*. *Chaotic iterations* have been introduced for this aim in [30].

Definition 5 *The* downward closure *of d by a set of operators R is $CL \downarrow (d, R) = \max\{d' \mid d' \subseteq d, \forall r \in R, d' \subseteq r(d')\}$.*

Note that if $R' \subseteq R$, then $CL \downarrow (d, R) \subseteq CL \downarrow (d, R')$.

Obviously, each solution to the CSP is in the downward closure. It is easy to check that $CL \downarrow (d, R)$ exists and can be obtained by iteration of the operator $d' \mapsto d' \cap \bigcap_{r \in R} r(d')$. Using *chaotic iteration* provides another way to compute $CL \downarrow (d, R)$ [22]. Iterations proceed by elementary steps. Chaotic iteration is a convenient theoretical definition but, in practice, each iteration is finite and fair in some sense.

Lemma 1 *The limit of every chaotic iteration of the set of local consistency operators R from $d \subseteq \mathbb{D}$ is the* downward closure *of d by R.*

This well-known result of confluence [22, 30] ensures that any chaotic iteration reaches the closure. Notice that, since $\subseteq$ is a well-founded ordering (i.e. $\mathbb{D}$ is a finite set), every iteration from $d \subseteq \mathbb{D}$ (obviously decreasing) is stationary, that is, $\exists i \in \mathbb{N}, \forall j \geq i, d^j = d^i$: in practice, computation ends when a common fix-point is reached (*e.g.* using a propagation queue).

1.3 Enumeration

The computation of a solution to a constraint problem often needs a so-called *enumeration* phase. Indeed, on many occasions, constraint propagation alone is not sufficient to reduce the environment to a set of singletons.

The enumeration phase can be modeled as a sequence of constraint additions and retractions (backtracks). As long as no solution is found, a variable with a domain with at least two values is selected and a *decision* is made: reducing the domain[2]. This reduction can be considered as the addition of a *decision constraint* to the current constraint system.

2. Classically, such a decision amounts to reducing the current domain to a single value (variable assignment) but this can be more general as in numeric CSP where a splitting is made, or for scheduling problems where a precedence constraint is posted.

Definition 6 *A* decision constraint *is a constraint that helps to reduce at least one domain in an environment when solving a constraint satisfaction problem.*

Then, a propagation step is performed. As usual, if a domain is emptied, a backtrack occurs (the last decision is retracted); if the current environment is not reduced to a set of singletons, the decision process is repeated until a solution is found or no decision constraint is left to be retracted (the problem is over-constrained). This is a tree-based search process.

As the constraints system evolves throughout resolution, a notion of *context* is needed to describe a given state of the resolution.

Definition 7 *A* context *for a constraint satisfaction problem is composed of two sets: the set of the original constraints of the problem and a set of decision constraints. Typically, the latter represents the current path in the search tree that is being explored.*

2. Explanations for constraint propagation

Informally, an *explanation-set* is a set of constraints that *justifies* a domain reduction. As we will see through the concept of *explanation-tree*, explanations can be derived from the resolution of the CSP.

2.1 The basics: explanation-sets

Definition 8 *Let R be the set of all local consistency operators. Let $h \in \mathbb{D}$ and $d \subseteq \mathbb{D}$. We call an* explanation-set *for h* w.r.t. *d a set of local consistency operators $E \subseteq R$ such that $h \notin CL \downarrow (d, E)$.*

Since $E \subseteq R$, $CL \downarrow (d, R) \subseteq CL \downarrow (d, E)$. Hence, if E is an explanation-set for h then each super-set of E is an explanation-set for h. An explanation-set E is independent of any chaotic iteration with respect to R.

For each $h \notin CL \downarrow (d, R)$, expl($h$) represents any explanation-set for h. Notice that for any $h \in CL \downarrow (d, R)$, expl($h$) does not exist.

2.2 More information: explanation-trees

Explanation-sets are a compact representation of a sufficient set of constraints to achieve a given domain reduction. A more complete description of the interaction of the constraints responsible for this domain reduction can be introduced through *explanation-trees*. We need to introduce the notion of deduction rule related to local consistency operators.

2.2.1 Deduction rules and local consistency operators

Definition 9 *A* deduction rule *of type (W_{in}, W_{out}) is a rule $h \leftarrow B$ such that $h \in \mathbb{D}|_{W_{out}}$ and $B \subseteq \mathbb{D}|_{W_{in}}$.*

The intended semantics of a deduction rule $h \leftarrow B$ can be presented as follows: if all the elements of B are removed from the environment, then h does not appear in any solution of the CSP and may be removed harmlessly. B is a *support set* for h.

A set of deduction rules $\mathcal{R}_r$ may be associated with each local consistency operator r. It is intuitively obvious that this is true for arc-consistency enforcement but it has been proved in [32] that for any local consistency which boils down to domain reduction it is possible to associate such a set of rules (moreover it shows that there exists a natural set of rules for classical local consistencies). It is important to note that, in the general case, there may exist several rules with the same head but different bodies.

We consider the set $\mathcal{R}$ of all the deduction rules for all the local consistency operators of R defined by $\mathcal{R} = \cup_{r \in R} \mathcal{R}_r$.

The initial environment must be taken into account in the set of deduction rules: the iteration starts from an environment $d \subseteq \mathbb{D}$; it is therefore necessary to add facts (deduction rules with an empty body) in order to directly deduce the elements of $\overline{d}$: let $\mathcal{R}^d = \{h \leftarrow \emptyset \mid h \in \overline{d}\}$ be this set.

2.2.2 Proof-trees

Definition 10 *A* proof-tree *with respect to a set of rules $\mathcal{R} \cup \mathcal{R}^d$ is a finite tree such that for each node labelled by h, let B be the set of labels of its children, $h \leftarrow B \in \mathcal{R} \cup \mathcal{R}^d$.*

Proof-trees are closely related to the computation of environment/domain reduction. Let $d = d^0, \ldots, d^i, \ldots$ be an iteration. For each i, if $h \notin d^i$ then h is the root of a proof-tree with respect to $\mathcal{R} \cup \mathcal{R}^d$. More generally, $\overline{CL \downarrow (d, R)}$ is the set of the roots of proof trees with respect to $\mathcal{R} \cup \mathcal{R}^d$.

Each deduction rule used in a proof-tree comes from a pack of deduction rules, either from a pack $\mathcal{R}_r$ defining a local consistency operator r, or from $\mathcal{R}^d$.

A set of local consistency operators can be associated with a proof-tree:

Definition 11 *Let t be a proof-tree. A* set X of local consistency operators associated with t *is such that, for each node of t: let h be the label of the node and B the set of labels of its children: either $h \notin d$ (and $B = \emptyset$); or there exists $r \in X, h \leftarrow B \in \mathcal{R}_r$.*

Note that there may be several sets associated with a proof-tree. Moreover, each super-set of a set associated with a proof-tree is also convenient (R is associated with all proof-trees). It is important to recall that the root of a proof-tree does not belong to the closure of the initial environment d by the set of local consistency operators R. So there exists an explanation-set (Definition 8) for this value.

Lemma 2 *If t is a proof-tree, then each set of local consistency operators associated with t is an explanation-set for the root of t.*

From now on, a proof tree with respect to $\mathcal{R} \cup \mathcal{R}^d$ is therefore called an *explanation-tree*. Lemma 2 shows *explanation-sets* can be computed from *explanation-trees*.

2.3 Basic properties of explanations

Characterizing explanations is helpful when comparing or using explanations. We introduce several concepts that will be used later in this document: preciseness, validity and k-relevance.

2.3.1 Preciseness

Definition 12 *An explanation-set e_1 is said to be* more precise *than explanation-set e_2 iff $e_1 \subset e_2$.*

This is a simple way of defining precise explanation-sets. However, there are other possibilities: preciseness could be defined regarding the reduction power of operators, the scope (number of involved variables) of the constraints, etc.

Notice that determining for any value removal the most precise explanation-set (at least one of them as several non-comparable ones may exist) often amounts to solving an NP-hard problem.

2.3.2 Validity and k-relevance

Explanation-sets are a self-contained concept. However, a given explanation-set may not be relevant in the current context (the set of constraints and active decisions).

Definition 13 *An explanation-set e is said to be* valid *or* relevant w.r.t. *the current context C iff*

$$\{c|\exists r \in e, r \in R(c)\} \subset C$$

Following [8], we introduce the concept of k-relevance for explanation-sets to measure their distance from full validity.

Definition 14 *An explanation-set e is said to be* k-relevant w.r.t. *the current context C iff*

$$\#\{c|\exists r \in e, r \in R(c), c \notin C\} < k$$

We have recently developed several tools based on k-relevance that are presented in [62].

2.4 Related concepts

Explanations, as presented here, generalize *justifications* from [11, 25] that were introduced to dynamically remove constraints. A justification for a value removal is represented by the constraint (or the propagation operator) that actually removed the value.

Conflict-sets (*a.k.a.* nogoods [72]) are also a specific case of our explanations. Indeed, a conflict-set is a set of assigned variables that is contradictory *w.r.t.* the constraints of the problem. An explanation-set can be derived from this conflict by considering the associated decision constraints.

Contradiction explanations for [46] for configuration problems are sets of original constraints for which propagation leads to at least one empty variable domain. They do not take into account decision constraints and therefore are limited to local consistency contradictions and do not encompass global consistency contradictions. Such a limitation allows the design of polynomial algorithms to determine explanations.

Explanations for [75, 33] are a user-related concept. User-relevant information is associated to each constraint in advance and provided to the user in case of a contradiction. Our approach is different in the sense that we would like to automatically generate explanations for any given constraint set without determining in advance the nature of the explanations to provide.

Summary

In this second part, we introduced a formal framework for constraint programming in which we defined the notion of explanation. We introduced two different representations of explanations: a detailed one (explanation-trees) and a compact one (explanation-sets). Moreover, we characterized explanations in order to discriminate different explanations for a same event. Finally, we related our explanations to previous concepts such as justifications, conflict-sets and contradiction explanations.

III. Computing explanations: techniques and implementation

This third part is dedicated to the different techniques used to compute explanations. Our own implementation is described.

3. Computing explanations

As we have seen earlier, explanation-sets are closely related to explanation-trees which are themselves closely related to the actual constraint propagation. More formally, we introduce a dynamic way of computing explanation-trees during propagation, first from a theoretical point of view and secondly from an operational point of view.

3.1 Using the formal definition

Let us consider a fixed iteration $d = d^0, d^1, \ldots, d^i, \ldots$ of R with respect to $r^1, r^2, \ldots$. In order to incrementally define explanation-trees during an iteration, let $(S^i)_{i\in\mathbb{N}}$ be the family recursively defined as:

- $S^0 = \{\text{cons}(h, \emptyset) \mid h \notin d\}$;
- $S^{i+1} = S^i \cup \{\text{cons}(h, T) \mid h \in d^i, T \subseteq S^i, h \leftarrow \{\text{root}(t) \mid t \in T\} \in \mathcal{R}_{r^{i+1}}\}$

where $\text{cons}(h, T)$ is the tree defined by h the label of its root and T the set of its subtrees, and where $\text{root}(cons(h, T)) = h$.

It is important to note that some explanation-trees do not correspond to any iteration, but when a value is removed there is always an explanation-tree in $\bigcup_i S^i$ for this value removal.

Among the explanation-sets associated with an explanation-tree $t \in S^i$, one is preferred. This explanation-set is denoted by $\text{expl}(t)$ and defined as follows (where $t = \text{cons}(h, T)$):

- **if** $t \in S^0$ **then** $\text{expl}(t) = \emptyset$;
- **else if** there exists $i > 0$ such that $t \in S^i \setminus S^{i-1}$, **then** $\text{expl}(t) = \{r^i\} \cup \bigcup_{t'\in T} \text{expl}(t')$.

In fact, $\text{expl}(t)$ is $\text{expl}(h)$ previously defined where t is rooted by h.

We will associate a single explanation-tree, and therefore a single explanation-set, to each element h removed during the computation. This set will be denoted by $\text{expl}(h)$.

3.2 Event-based finite domain constraint solvers

We consider here constraint solvers that manipulate finite domain variables and constraints implemented as local consistency operators as presented above. The computation of the greatest common fix-point for a set of constraints (operators) is done through an event-based system: operators are awakened/called each time a variable domain is reduced (this is an *event*). Those same operator calls can (and should) themselves generate new events[3].

In such a constraint solver, a constraint is fully characterized by the set of local consistency operators to be applied upon an event: a value removal, a domain bound update, a variable instantiation, a set of value removals, etc. In the following, we will focus our presentation on the first two events since all the others can be expressed in terms of these first two.

3. As recalled in [59], this architecture is well suited to solvers that maintain a local consistency (arc-consistency and others) but is not so well suited to consistencies that are defined upon several constraints at the same time.

Many existing solvers fit this description: PROLOG based solvers (*e.g.* ECLIPSE, SICSTUS, etc.), C++ based solvers (*e.g.* ILOG SOLVER, FIGARO, CHIP++), JAVA based solvers (*e.g.* ILOG JSOLVER) and CLAIRE based solvers (*e.g.* CHOCO).

In the following, we will illustrate implementation concepts with CHOCO. Local consistency operators for a given constraint are implemented in CHOCO using a specific method for each handled event (possibly referring to the same generic method): `awakeOnRem` for a value removal, `awakeOnInf` for a lower bound modification and `awakeOnSup` for an upper bound modification. Example 1 describes the `awakeOnSup` method for constraint $x \geq y + c$.

3.3 From theory to practice

The previous formal presentation of an interactive computation of explanations during constraint propagation (an iteration) can easily be used in an actual implementation. Obviously, in constraint solvers, deduction rules are often not explicitly stated in the constraint propagation code. However, their effect is always implemented: it is the actual domain reduction. Therefore, an obvious way to implement an explanation-based constraint system is to enhance the constraint propagation code with the deduction rules that are being applied.

For example, if we consider constraint $x \geq y+c$ as presented in Example 1, the proper position to compute an explanation for a domain modification (here an update on the upper bound of a variable) is when actually calling the `updateSup` method (and not within that same method, because we would have lost the actual reason why the constraint performed the domain modification). Section 4 presents how to actually compute explanations in that position. More generally, an explanation is to be computed and stored each time the solver performs a domain reduction.

In the following, for the sake of clarity, we will choose to manipulate not explanation-trees but explanation-sets (deduced from the explanation-trees) and to manipulate directly constraints instead of operators. Therefore, all explanations that will be used will simply be a set of constraints.

4. Implementing an explanation-based constraint solver

In this section, we introduce the PALM system: our explanation-based constraint solver built on top of CHOCO. Notice that what we present here remains valid for any event-based finite domain constraint solver.

4.1 Introducing explanations within an existing solver

Explanation-trees are collapsed into explanation-sets, themselves collapsed into a set of constraints (explanations for short) instead of sets of local consistency operators. Moreover, we only consider a single explanation for a given event.

4.1.1 Storing and retrieving explanations

Explanations are stored in the following way:

- for variables represented by their lower and upper bound, two stacks of explanations are stored: one for each bound.
- for variables represented by the set of all possible values, an explanation is associated with each value.

In the following, we will use a general explanation building method: `becauseOf` which builds up an explanation by joining explanations associated with a list of event-related information that is provided as a parameter. The following parameters are available:

- `theSup(x)` (resp. `theInf(x)`) for explaining the current upper (resp. lower) bound of variable x. If the variable domain is represented by its two bounds, this amounts to checking the last entry in the associated stack; if the variable domain is represented as a set of possibles values, the union of the explanation for the values above (resp. below) the current bound is returned.

- `theDom(x)` for explaining the current domain of variable x. This amounts to joining the explanations for all the value removals (or for the two current bounds) for the variable.

- `theHole(x,v)` for explaining the removal of value v from x.

- `theConstraint(c)` for adding a given constraint to the computed explanation.

4.1.2 Instrumenting the propagation code

Explanations for events need to be computed and attached when the events are generated *i.e.* within the propagation code of the constraints (namely the `awakeOnXXX` methods of CHOCO). We therefore need to add extra information to the `updateInf`, `updateSup` or `removeVal` calls: the actual explanation. Example 2 shows how such an explanation can be computed and what the resulting code is for a basic constraint.

> **Example 2 (Modifying the solver):**
> It is quite simple to make modifications considering Example 1. Indeed, all the information is at hand in the `awakeOnSup` method. The modification of the upper bound of variable `c.v2` (y) is due to:
>
> (a) the use of the constraint itself (it will be added to the computed explanation);
>
> (b) the previous modification of the upper bound of variable `c.v1` (x) that we captured through the *calling* variable (`idx`).
>
> The source code is therefore modified in the following way (the additional third parameter for `updateSup` contains the explanation attached to the intended modification):
>
> ```
> [awakeOnSup(c:GreaterOrEqualxyc,idx:integer)
>  -> if (idx = 1)
>       updateSup(c.v2, c.v1.sup - c.cste, becauseOf(theConstraint(c), theSup(c.v1)))]
> ```

More generally, if we consider explanations for classical binary CSP (arc-consistency enforcement) where constraints are expressed by a list of allowed tuples of values for their variables (*i.e.* given in *extension*), when applying constraint c_{xy} between variables x and y, we want to remove value a from the domain of x if and only if all supporting values for a in the domain of y regarding constraint c_{xy} have been removed. This can be expressed this way:

$$\texttt{expl}(x \neq a) = c_{xy} \bigcup_{b \ \texttt{supp.}\ a} \texttt{expl}(y \neq b)$$

These modifications added to each local consistency rapidly give an *explanation-friendly* constraint solver.

4.2 Handling global constraints

Global constraints are now widely used within constraint programming. They are often used to encompass in a single constraint the resolution of a whole complex problem: `allDifferent` [68] solves a matching problem between a set of variables and a set of values, `cumulative` [1] solves a cumulative problem on a given resource, `stretch` [63] for solving a shift assignment problem, etc.

4.2.1 Global constraints and explanations

Although providing precise explanations is quite easy for basic constraints because the existing code explicitly states what the real triggering conditions are (as in Example 2), instrumenting global constraints is not so easy. Indeed, as some complex data structures are maintained, value removals occur when an accumulation of information modifies them enough to trigger some specific rules. Example 3 roughly describes the behavior of the `allDifferent` constraint and the problem of computing precise explanations.

> **Example 3 (The allDifferent constraint):**
> The `allDifferent` constraint [68] aims at ensuring that there is a maximal matching between variables and their values (*i.e.* it is possible to assign all variables to different values). It is handled using graph theory results. In a few words, the maintained data structure allows the removal of a value from the domain of a variable as soon as these two (variable and value) appear in different strongly connected components in a specific graph updated throughout the computation. What would be an explanation for such a value removal ? If no information is kept or no rewriting is done, the only information would be that the removal is due to the current state of all the variables that appear in the constraint. This is not really precise information. However, analyzing the algorithm itself makes it possible to provide more precise information (*i.e.* restrict the explanation) [2].

When triggering the propagation rule, not all of these pieces of information are kept. There is always a generic explanation: the current state of the domains of each variable of the constraints. Such an explanation is useless if used with all the constraints of the system because everything will depend on everything else!

When adding explanations to global constraints, it is highly recommended to get back to the overall algorithm in order to make it *explanation-friendly i.e.* implemented in such a way that computation of precise explanations will be easy. Moreover, notice that such an explanation-friendly algorithm can act as self-documentation for the constraint. This is illustrated in a recent work presented in [70] where several global constraint are studied in order to provide precise explanations.

4.2.2 An example: scheduling-related constraints

For solving scheduling problems, one of the key techniques used is called *immediate selection*. It is a domain reduction technique for unary resource constraints [19]. The main idea of *immediate selection* is to identify a task t and a set S of tasks that share a common unary resource such that it can be proven that t cannot be scheduled before all the tasks in set S. Then, the lower bound for the starting time of task t can be modified in order to reflect that new information. There are (at least) two implementations of *immediate selection*:

- one [19] does not explicitly compute set S but only the adjustment that can be made to the starting time of task t: it is not explanation-friendly. There is no way of precisely explaining the adjustments.
- another one [20] uses another point of view. The idea is to maintain a set of tasks (prospective sets S) that fit in a given evolutive interval of time (namely *task-intervals*) and to check whether a candidate task t exists for adjustments. This is an equally efficient technique as the first one but it is clearly explanation-friendly. Indeed, an explanation for any adjustment can be restricted to explanations of the current domains of the tasks in set S that is here explicitly available. [52] shows the interest of such a technique for explanations.

4.3 Complexity issues

One of the major interests of our approach is that both space and time complexities remain polynomial. As the explanation network *traces* the behavior of the solver, when a single value is removed

(which happens at most once for each possible value) a single (but still usable) explanation will be generated. Thus, the space complexity for a CSP defined upon n discrete variables of maximum domain size d upon which are posted e constraints is in $O((e+n) \times n \times d)$. There are at most $n \times d$ explanations (one for each value) of maximum size e (all the constraints of the problem) $+n$ (one decision[4] constraint for each variable).

Experimental results presented in [51, 69] show that computing explanations while filtering lead to a time overhead due to actual computing and storing of explanations, but this overhead remains quite constant whatever the nature of the problem (highly constrained problems *vs.* under-constrained problems). Moreover, this overhead is rewarded when we actively use these explanations for, for example, guiding the search as we will see in Section 8.

4.4 The PALM system

PALM[5] is our explanation-based constraint solver [48]. It is provided as a free CHOCO library. It implements concepts and algorithms presented here.

The main feature of PALM is that it provides tools to explicitly compute, store and retrieve explanations for every domain modification in a given problem. But, it is also a constraint solver:

- PALM is a **classic** constraint solver: it can be used to solve problems as if it were CHOCO.
- PALM is a **dynamic** constraint solver: it handles dynamic addition and retraction of constraints before, during or after resolution (see why in Section 7.1).
- PALM is an **explanation-based** constraint solver: it provides specific search algorithms that make an active use of explanations (see why in Section 8).

In the following, we will provide examples and experiments using the PALM system.

5. Other computation techniques

Our approach amounts to *tracing* a constraint solver without modifying the classical mechanisms of constraint programming (filtering and enumeration). Our computed explanations therefore cannot be guaranteed to be minimal (although experiments show that they remain quite precise) nor exhaustive (other explanations may exist). Moreover, they are highly dependent on the solver mechanisms and implementation. It is a pragmatic point of view about explanation/conflict computation.

There are alternatives:

- *off-line* techniques: computing all possible explanations before computation or computing on demand explanations only when a contradiction has been identified.
- *non-intrusive* techniques: computing explanations during computation without modifying the existing solver.

5.1 Off-line techniques

There are several off-line techniques to compute explanations. In these techniques, computing explanations is completely independent from solving the actual problem. *A priori* techniques compute sets of explanations before solving a problem in order to instantaneously answer user queries after resolution has terminated. *A posteriori* techniques aim at quickly computing relevant information upon a user query.

4. We consider here decision constraints as assignment constraints. There are at most n such constraints in any given context: only one value can be assigned to a single variable.
5. e-constraints.net

5.1.1 A priori techniques

[75] introduce explanations while solving logic puzzles. The idea is to use an inference-based solver that generates a *pre-compiled* explanation for each event.
[5] are interested in the design of interactive and diagnosis tools in decision support systems for configuration problems. There are two main differences with our explanations:

- the CSP representing the initial problem is considered to be persistent and compiled into an automaton;
- as a consequence, a user can only interact by retracting/adding decision constraints.

Our explanations are not restricted to decision constraints.

5.1.2 A posteriori techniques

QuickXPlain [46] following work from [24] and [7] iteratively tests the local consistency of subsets of the constraints to compute a minimal conflict for a given inconsistent set of constraints following the ideas of the QuickSort algorithm. QuickXPlain is therefore a polynomial algorithm that computes a subset of existing explanations for contradictions (see Section 2.4).

5.2 Non-intrusive techniques

Non-intrusive techniques do not need to modify existing propagation code in order to provide explanations. Obviously off-line techniques are non-intrusive techniques. However, we are currently investigating new techniques that can be used to monitor a constraint solver from the outside while it runs and still provide precise explanations for its behavior. These techniques are based on a recent tool: Aspect Oriented Programming (AOP) [57].

5.2.1 Aspect Oriented Programming

Separation of concerns is a well-known topic in software engineering. For instance, the Apache web server[6] can be decomposed at the design level into a series of concerns: XML parsing, URL pattern matching, session logging, etc. At the code level, it is often impossible, when a system is complex, to find an architecture that expresses the different concerns in a modular way. For example, in the Apache web server there is no module for session logging. Indeed, the corresponding code is distributed in many places in the other modules as shown on the slide[7] in Figure 1. Each vertical white bar represents a module, and each horizontal gray line in a white bar represents a line of code implementing session logging.

This slide can be used to describe the explanation computation code within a constraint solver. Computing explanations as presented above amounts to systematically modifying existing solvers in order to add the explanation-related instructions.

Current constraint programming tools and languages do not support crosscutting concerns, and the programmer must tediously insert the explanation computation code by hand in all modules (see Section 4.1). Moreover, the lack of modularity of such a crosscutting concern makes its maintenance difficult. AOP [57] is a new programming paradigm supporting crosscutting concerns. It allows the programmer to define the different concerns in a modular way; and it provides the programmer with a weaver responsible for mixing the different pieces of code together to generate the complete application.

Aspect-J[8] [56] is an extension of Java that supports AOP. First, a base program is written in Java, and several aspects are defined in Aspect-J. Second, the aspect-weaver, basically (but not only)

6. apache.org
7. O'Reilly Conference on Enterprise Java, March 29, 2001, `pdf` and `ppt` available at aspectj.org/doc/papersAndSlides.
8. aspectj.org

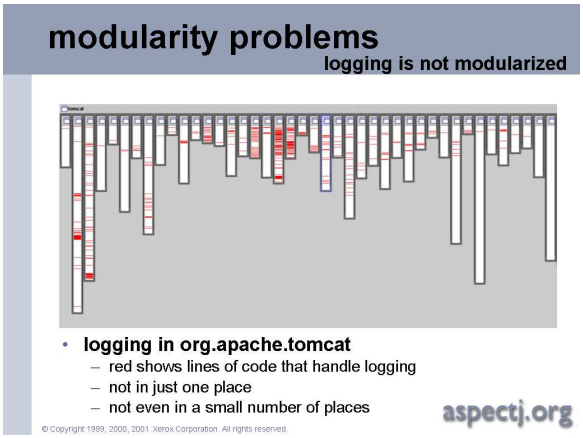

Figure 1: In Apache, session logging *crosscuts* the other concerns.

a preprocessor[9], inserts calls to the aspectual code into the base program to generate the complete program.

5.2.2 First steps with AOP for computing explanations

In order to validate the use of AOP in an explanation computation context, we presented in [27] the addition of explanation capabilities into a toy solver designed in Java. This minimal solver was meant to illustrate the basic concepts of constraint solvers: constraints implemented as propagators, a propagation queue, search performed through enumeration, etc. (as in Section 3.2).

We designed several aspects for: computing explanations, using explanations to help search, etc. This successful use of AOP leaves many interesting open points: how to compute precise explanations for global constraints from the outside; how to design explanation-related aspects in existing solvers; etc.

The main point to address is, in our opinion, that Aspect-J (and more generally AOP) is meant to access the support language of programs. Constraint programming has many identified concepts that one would like to be able to refer to when designing aspects (*e.g.* variable removing, backtrack, etc.) but that are not easily accessible in the support language (being implemented using several lines of code). We strongly believe that a way of overcoming this issue is to design an Aspect-Constraint language that would only manipulate constraint programming concepts.

Summary

This third part was dedicated to the computation of explanations. We showed how the theoretical framework introduced in the previous part could be implemented in an existing constraint solver. We emphasized the issues of adding explanation capabilities to global constraints. Finally, we introduced other computation techniques: off-line approaches and non-intrusive techniques. For the latter, we introduced the promising use of aspect-oriented programming.

9. However, it is believed that its semantics are easier to understand from a monitoring perspective [28].

IV. Applications

This fourth part is dedicated to the many possible uses of explanations: providing user information, incrementally handling dynamic problems and designing new conflict-based search techniques. A case study is detailed: solving open-shop scheduling problems.

6. Explanations for user interaction

Using constraint solvers or developing constraint-based systems may be a tedious and frustrating task. A classical way of exploiting trace-like information (such as explanations) is to use them as analysis tools.

6.1 Explanation-related queries

Using an explanation-based constraint solver helps to answer some classical queries when looking at a solution (or a no-solution message) for a CSP.

- *Why does my problem have no solution ?*
  When the domain of a variable v is empty (no value exists that will respect all the constraints on that variable), constraint systems usually notify users that there is no solution. Checking the explanation for this situation (*i.e.* computing $\bigcup_{a \in v}$ `expl`$(v \neq a)$) helps to understand why this is so by narrowing the problem to relevant constraints. In PALM, the command `becauseOf(theDom(`v`))`, if v is the emptied variable, will give such an explanation.

- *Why cannot variable x take the value a ?*
  Checking the associated removal explanation (*i.e.* `expl(`$x \neq a$`)`) helps to answer the question. In PALM, we simply use the command `becauseOf(theHole(`x`, `a`))`.

- *What if constraint c gets back into the system ?*
  In this situation, constraint c has been retracted. Keeping track of explanations that contained constraint c before its retraction will allow a glimpse (a necessarily partial but still valid one) of the effects of the re-activation of c. Indeed, events attached to those explanations would become valid. k-relevant explanations is a way of keeping track of past explanations [62].

6.2 Application: the PTIDEJ system

We experimented the usage of explanations in a debugging and analysis context in the PTIDEJ system [3]. PTIDEJ (Pattern Trace Identification, Detection and Enhancement in Java) is an automated system for identifying distorted micro-architectures in object-oriented source code.

The production of quality source code is an important issue for the software industry. A quality source code facilitates evolutions and maintenance: addition of new functionalities, bug corrections, adaptation to new platforms, integration with new class libraries. In object-oriented programming, a quality source code has two aspects: efficient and clearly-written algorithms, respecting conventions and idioms, and an *elegant* class architecture. A micro-architecture describes the structure of a subset of the classes of an object-oriented program. The solutions proposed by *design patterns* [34] are examples of **good** micro-architectures: they capture the experience of skillful developers.

However, it is not easy to write directly source code that carefully respects design patterns. Thus, most of the time, only *distorted* design patterns are present in the source code (*i.e.* micro-architectures similar to – not identical to – those proposed by the design patterns). There is a lack of

tools helping to identify distorted versions of design patterns in existing source code and indicating possible improvements by pointing out differences with the *exact* design pattern.

The PTIDEJ system is such a tool. Its aim is to identify distorted micro-architectures in object source code. The idea of PTIDEJ is to define a CSP whose structure (variables and constraints) represents the wanted design patterns and whose semantics (domains and constraint semantics) are derived from the object source code. Finding solutions to this CSP gives exact matching micro-architectures in the code. For finding distorted ones, PTIDEJ uses an explanation-based constraint solver to determine why the exact structure cannot be found and to guide the user towards the removal of unverified constraints.

PTIDEJ is the first existing tool able to identify distorted micro-architectures [42]. Moreover, it is able to explain why the answer it gives is indeed a distorted design-pattern. This is quite important because coding is often considered as an art and therefore automatic systems are often not well received by developers.

6.3 Open issues: user accessibility

As they are defined, explanations are sets of local consistency operators (or, in the applications section, low-level constraints). Only a specialist (or the system itself) can understand and correctly interpret the provided information[10]. When dealing with real users, we cannot use the explanations as they are. We need to provide *translation* tools to make the low-level constraints accessible to any user.

To address this issue, we introduced the use of user-friendly explanations [55]. The idea is to use the expertise of the developer of the final application. Indeed, when developing an application, such an expert needs to *translate* the problem from the high-level representation (the user's comprehension of the problem) to the low-level representation (the actual constraints in the system). This is a `user` → `system` translation. To interact with the user, we need the reverse *translation*: from the low-level constraints (solver-adapted) to the user-comprehensible constraints (higher level of abstraction). We note this translation a `system` → `user` translation. It is usually not explicitly coded in the system. Asking a developer to provide such a translator while coding would be quite strange for him. We chose to automatize this translation in a transparent way.

This idea relies on a single hypothesis: all aspects of a constraint-based application can be represented in a hierarchical way. Indeed, any object appearing in a constraint problem is attached to, at most, a single *father*-object. Note that a given *father*-object may have several *children*-objects. For example, Figure 2 gives a graphical representation of a conference organization problem described in Example 4 and modeled in Example 5.

10. For example, $x + 3 \geq y$ is not informative, however the expression it comes from, *task x with duration 3 must precede task y*, is more understandable.

Example 4 (A conference problem):
Michael, Peter and Alan are organizing a two-day seminar to write a report on their work. In order to be efficient, Peter and Alan need to present their work to Michael and Michael needs to present his work to Alan and Peter (actually Peter and Alan work in the same lab). Those presentations are scheduled for a whole half-day each. Michael wants to know what Peter and Alan have done before presenting his own work. Moreover, Michael would prefer not to come in the afternoon of the second day because he has a very long ride home. Finally, Michael would really prefer not to present his work to Peter and Alan at the same time.

Example 5 (A constraint model for a conference problem):
A constraint model for this problem is described as follows: let Ma, Mp, Am, Pm be the variables representing four presentations (M and m are respectively for Michael as a speaker and as a listener). Their domain will be $[1, 2, 3, 4]$ (1 is for the morning of the first day and 4 for the afternoon of the second day). Several constraints are contained in the problem: implicit constraints regarding the organization of presentations and the constraints expressed by Michael.
The implicit constraints are: *a speaker cannot be a listener in the same half-day* (*i.e.* $c_1 : Ma \neq Am$, $c_2 : Mp \neq Pm$, $c_3 : Ma \neq Pm$ and $c_4 : Mp \neq Am$) and *no one can attend two presentations at the same time* (*i.e.* $c_5 : Am \neq Pm$).
Michael's constraints are: *he wants to speak after Peter and Alan* ($c_6 : Ma > Am$, $c_7 : Ma > Pm$, $c_8 : Mp > Am$ and $c_9 : Mp > Pm$); *he does not want to come on the fourth half-day* ($c_{10} : Ma \neq 4$, $c_{11} : Mp \neq 4$, $c_{12} : Am \neq 4$ and $c_{13} : Pm \neq 4$) and *he does not want to present to Peter and Alan at the same time* ($c_{14} : Ma \neq Mp$).

While developing a constraint application, the user only needs to explicitly state the underlying hierarchy of his problem. Only the leaves of this structure, namely the low-level constraints, can be used by the constraint solver. But, as we have seen, the leaves may be too low-level for a typical user of the final application. However, he can understand higher levels in the hierarchy. What allows the hierarchy hypothesis is the building, with no effort for the developer, of a hierarchical representation of the problem. Once built, this representation can be used to interact with any user through user-friendly explanations. Such explanations are provided using procedures converting low-level constraints into user-understandable pieces of the hierarchy. These procedures are completely problem-independent and may be provided within the constraint solver.

The user perception of a given problem can be seen as a *cut* in the hierarchical view of the considered problem. For example, let us suppose that the user is Michael who does not want to deal with implicit constraints, although he does understand his own wishes. Therefore, his view of the problem would be: The conf. problem, P&A before, Not 4^{th} 1/2 day and P&A not same time.

Our example has no solution, one explanation for that situation is: $\{c_1, c_2, c_3, c_4, c_5, c_6\}$. If Michael looks at the explanation from his point of view, we need to project the concrete explanation onto his representation. The projection consists in projecting the constraints $\{c_1, c_2, c_3, c_4, c_5, c_6\}$ from the bottom to the top of the hierarchy representing the problem (see Figure 2) until a user-understandable box is reached.

For example, the projection of the constraints $\{c_1, c_2, c_3, c_4\}$ gives at first the Speaker vs. Listener box. Unfortunately, this box is not understandable by Michael. In this case, the projection continues to the *father* box: Implicit constraints. Once again, this box is not understandable by Michael and the projection gets to The conf. problem. Finally, we reach The conf. problem box which Michael can deal with. The projection of c_5 gives the same box through Listener vs. 2 pers. and Implicit constraints. For c_6, the projection is easier because the first box reached is user-understandable. The final projection gives: The conf. problem and P&A before.

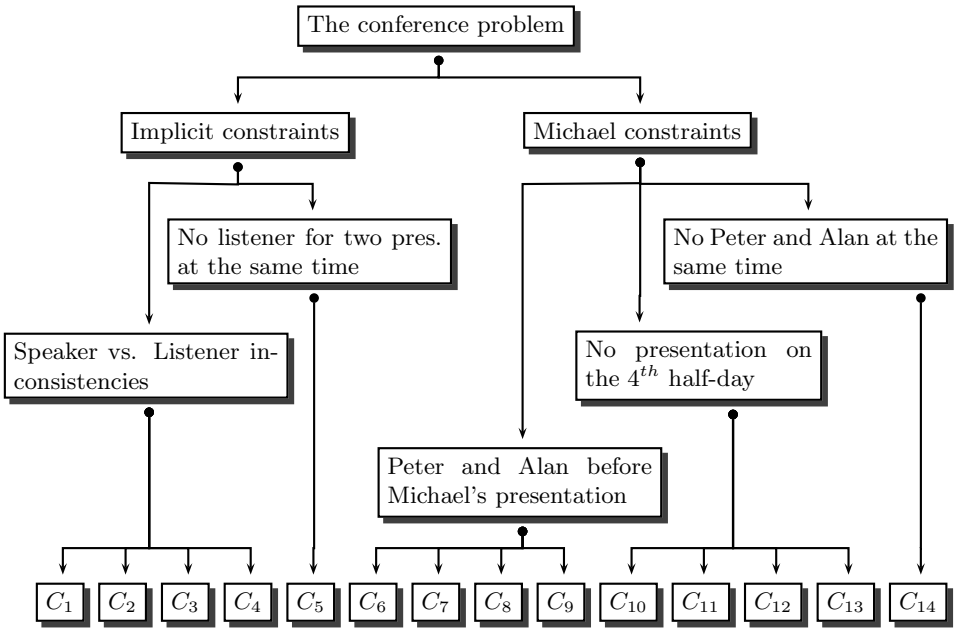

Figure 2: A hierarchical view of a conference problem

User-friendly boxes are implemented in the PALM system. It is only a first step towards a user-friendly explanations. The hierarchical hypothesis is to be confirmed[11], interactive tools need to be designed, etc.

7. Explanations for incremental handling of dynamic problems

Incremental constraint addition to a problem is a well-known issue in classical constraint programming solvers: it is often the usual way constraints are added to the constraint system. However, incremental constraint *retraction* is not so easy. Several extensions have been proposed to handle dynamic retraction of constraints: some of them [10, 35] analyze the reduction operators to be able to determine the past effect of a constraint and so incrementally retract it; others, following [11], store information to achieve that determination.

Explanations (or simplifications) may be used as past effect determination tools (see for example [25, 49, 50]). We detail in this section how incremental constraint retraction can be achieved using explanations.

7.1 Incremental constraint retraction

Let us consider a finite iteration from an initial environment d with respect to a set of operators R. At the step i of this iteration, the computation is stopped. The current environment is d^i. Note that this environment is not necessarily the closure of d by R (we have $CL \downarrow (d, R) \subseteq d^i \subseteq d$). At this i^{th} step of the computation, some constraints have to be retracted. Performing constraint retraction amounts to [35, 47]:

1. **Disconnecting** The first step is to cut the retracted constraint c from the constraint network. c needs to be completely disconnected (and therefore will never get propagated again in the future). Disconnecting a constraint c amounts to removing all its related operators from the current set of active operators. The resulting set of operators is $R^{new} \subseteq R$, where $R^{new} = \bigcup_{c' \in C \setminus \{c\}} R(c')$ where $R(c')$ is the set of local consistency operators associated with c'. Constraint retraction amounts to computing the closure of d by R^{new}.

2. **Setting back values** The second step is to undo the past effects of the constraints, both the direct (each time the constraint operators have been applied) and indirect (further consequences of the constraint through operators of other constraints) effects of that constraint. This step results in the enlargement of the environment: values are put back.

   Here, we want to benefit from the previous computation of d^i instead of starting a new iteration from d. Thanks to explanation-sets, we know the values of $d \backslash d^i$ that have been removed because of a retracted operator (that is an operator from $R \setminus R^{new}$). This set of values is defined by $d' = \{h \in d \mid \exists r \in R \setminus R^{new}, r \in expl(h)\}$ and must be re-introduced into the domain. Notice that all incremental algorithms for constraint retraction amount to computing a (often strict) super-set of this set. We have proven that we obtain the same closure if the computation starts from d or from $d^i \cup d'$. Moreover, any constraint retraction algorithm that re-introduces a super-set of d' remains correct [26].

3. **Controlling what has been done** Some of the values put back can be removed applying other active operators (*i.e.* operators associated with non retracted constraints). These environment reductions need to be performed and **propagated** as usual.

   In practice, the iteration is done with respect to a sequence of operators, which is dynamically computed thanks to a propagation queue. At the i^{th} step, before setting values back, the set of operators that are in the propagation queue is R^i. Obviously, the operators of $R^i \cap R^{new}$ must

11. Although we did not find any problem that would not verify this assumption.

stay in the propagation queue. The other operators ($R^{new} \setminus R^i$) cannot remove any element of d^i, but they may remove an element of d' (the set of re-introduced values). So we have to put some of them back in the propagation queue: the operators of the set $R' = \{r \in R^{new} \mid \exists h \leftarrow B \in \mathcal{R}_r, h \in d'\}$. We have proven that the operators that are not in $R^i \cup R'$ do not modify the environment $d^i \cup d'$, so it is useless to put them back into the propagation queue. Moreover, any constraint retraction algorithm that re-introduces a super-set of R' in the propagation queue remains correct [26].

At the end of this process, the system is in a consistent state. It is exactly the state that would have been obtained if the retracted constraint (c) had not been introduced into the system[12].

7.2 Efficient re-propagation of constraints

Although determining the past effect of a constraint c is quite easily done by referring to the explanations that do contain any reference to c, efficient re-propagation is not provided in classical constraint solvers. Indeed, another event needs to be handled: value restoration (or bound restoration). Instead of calling a general local consistency operator for handling this new event, it is possible to design new operators dedicated to this re-propagation phase *i.e.* to confirm the validity of value restorations: methods `awakeOnRestoreInf`, `awakeOnRestoreSup` and `awakeOnRestoreVal`.

Example 6 shows what such methods look like in PALM.

> **Example 6 (Value restoration for $x >= y + c$):**
> Let us consider a `GreaterOrEqualxyc` constraint. When the upper bound of the second variable of such a constraint is increased (this is a value restoration), we need to check if that new value is compatible with the constraint. An easy way to achieve this behavior is simply to pretend that the upper bound of the first variable has been modified, which gives the following code:
>
> ```
> [awakeOnRestoreSup(c: GreaterOrEqualxyc, idx: integer)
>  -> if (idx = 2) awakeOnSup(c, 1)]
> ```
>
> Conversely, if the lower bound of the first variable is restored we can use the following code:
>
> ```
> [awakeOnRestoreInf(c: GreaterOrEqualxyc, idx: integer)
>  -> if (idx = 1) awakeOnInf(c, 2)]
> ```
>
> Like in Example 1, `idx` is the index of the variable whose upper (*resp.* lower) bound has been restored.

More generally, `awakeOnRestoreXXX` methods should call (portions of) `awakeOnXXX` methods that would modify the part of the domain concerned by the event called with the method.

In an explanation-based solver, a constraint is therefore characterized by its reaction to all its domain modification events: value removals (`awakeOnInf`, `awakeOnSup`, and `awakeOnRem` methods) **and** value restorations (`awakeOnRestoreInf`, `awakeOnRestoreSup`, and `awakeOnRestoreVal` methods).

7.3 Over-constrained problems

We can use the same tool (explanation-sets) for both computing explanations for conflicts and for incrementally removing a constraint from a given constraint system. This naturally leads to considering using explanations for solving over-constrained problems.

We introduced in [50] a simple strategy: solve the complete constraint system *as usual* and, if necessary (*i.e.* once the problem has been proven over-constrained), use the conflict explanation to identify the next constraint system to consider (using a *comparator* [80] that takes into account

12. Notice that the state of the domain is the state that would have been obtained if the constraint was not originally in the system. But, the set of current explanations is only one of the several different states that are associated to the initial set of constraints without c.

the user's preferences) and perform the constraint modification (retraction(s) and/or addition(s)) incrementally still using explanations. This process is iterated as long as no suitable solution is found. The use of the comparator guarantees the optimality of the solution. Moreover, operational semantics in terms of tree search modification can be defined [50].

The originality of this approach is two-fold:

- all constraints of the problem can be propagated from the beginning of the search. The use of comparison criteria can be delayed until the last moment [50]. Indeed, choice criteria (reflecting the user opinion about relaxing constraints) are used only for selecting a set of constraints from among the ones appearing in the current conflict. Thus, classifying constraints is only needed when a contradiction is encountered and only between relevant constraints. We therefore do not need to provide user's advice on all the constraints of the system but instead we can design an interactive interrogation of the user when needed.

- search is made in the space of possible relaxations in opposition to the space of possible affectations in more classical approaches ([15] or [73]). Indeed, we perform search by dynamically adding or retracting constraints until an optimal configuration is reached. [15] uses a hierarchy of constraints introduced constraints level by level (from the more important to the less important ones) and [73] performs search by modifying complete assignments using constraint violations as guides for the search.

If we consider the classical enumeration process used to ultimately solve CSP as a dynamic sequence of constraint additions (assigning variables) and retractions (backtracks), we can see that the method that we introduced for solving over-constrained problems can be adapted to replace that enumeration process. In Section 8 we introduce this point of view in what we call *explanation-based constraint programming*.

7.4 Application: solving dynamic RCPSP

The Resource Constrained Project Scheduling Problem (RCPSP) is a general scheduling problem. It consists of a set of activities and a set of renewable resources. Each resource is available in a given constant amount. Each activity has a duration and requires a constant amount of resource to be processed. Preemption is not allowed. Activities are related by two sets of constraints: temporal constraints modeled through precedence constraints, and resource constraints that state that for each time period and for each resource, the total demand cannot exceed the resource capacity. The objective considered here is the minimization of the makespan (total duration) of the project. This problem is *NP-hard* [13].

Most work about RCPSP considers static problems in which activities are known in advance and constraints are fixed. However, every schedule is subject to unexpected events (consider for example a new activity to schedule, or a resource failure *e.g.* machine breakdown). When such a situation arises, a new solution, taking these events into account, is needed in a preferably short time. Two classical methods used to solve such problems are: recomputing a new schedule from scratch each time an event occurs (a quite time-consuming technique) and constructing a partial schedule and completing it progressively as time goes by (like in on-line scheduling problems – this is not compatible with planning purposes).

7.4.1 Explaining scheduling global constraints

CSP are increasingly used for solving scheduling problems. Many global temporal and resource constraints have been developed for solving such problems [21, 20, 58]. In [29] we introduce our application of explanation-based constraint programming to solve dynamic RCPSP. In order to use the capabilities of explanation for handling dynamic constraint addition and retraction, we need to add explanations to the specific constraints used for solving scheduling problems.

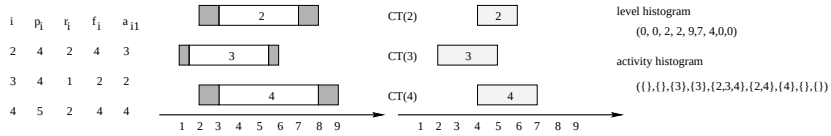

Figure 3: Example of timetable.

For example, we used the *resource-histogram* technique [21, 20] whose principle is to associate to each resource k an array $\texttt{level}(k)$ in order to keep a timetable of the resource requirements. This histogram is used for detecting a contradiction and reducing the time-window of activities.

The *core-times* technique [58] is used for computing lower bounds using a destructive method. A *core-time* $\texttt{CT}(i)$ is associated to each activity i. It is defined as the interval of time during which a portion of an activity is always executed whether it starts at its earliest or latest starting time. The lower bound of the schedule is obtained when considering only the *core-time* of each activity. If a resource-conflict is detected then some lower bounds can be upgraded.

Combining these two techniques provides an efficient resource-conflict detecting constraint. A timetable for each resource is computed as follows: for each activity i, its *core-time* $\texttt{CT}(i) = [f_i, r_i+p_i)$ (f_i is the latest starting time of i, r_i its earliest starting time and p_i is the duration of task i) is computed and the amount a_{ik} of resource k for each time interval $[f_i, f_i+1), \ldots, [r_i+p_i-1, r_i+p_i)$ is reserved. We associate to each *timetable constraint* two histograms (see Fig. 3):

- *a level histogram* which contains the amount of resource required at each time interval $[t-1, t)$.
- *an activity histogram* which contains the sets S_t of activities which require any amount of resource for each time period $[t-1, t)$. It will essentially be used for providing explanations.

These histograms are used in the following ways:

- detecting resource conflicts when the required level of a resource k at one time period t exceeds the resource capacity. The explanation-set associated to this contradictory situation is constituted from the set of activities (S_t) stored in slot t of the activity histogram. We obtain (c being the histogram constraint itself): $\left(\bigwedge_{v \in S_t} \left(\bigwedge_{a \in d(v)} \texttt{expl}(v \neq a)\right)\right) \bigwedge c$.
- tightening the time-window of an activity. It may occur that the current bounds of the time-window of an activity are not compatible with the other activities. In that situation, the tightening of the time-window will be explained by the set S_t of activities requiring the resources during the incompatible time-period $[t-1, t)$. We obtain (c being the histogram constraint itself): $\left(\bigwedge_{v \in S_t} \left(\bigwedge_{a \in d(v)} \texttt{expl}(v \neq a)\right)\right) \bigwedge c$.

.

7.4.2 An interactive system

We developed a branch-and-bound search algorithm using explanation-based techniques (see Section 8 for the general philosophy of this algorithm) based upon a branch-and-bound from [18]. This algorithm is the core part of an interactive RCPSP solving system.

This interactive system accepts several types of modification of the scheduling problem: temporal events (adding/removing precedence/overlapping/disjunctive relations, modifying time-windows), activity related events (adding/removing), resource related events (adding/removing/modifying). Each particular event is handled as a series of incremental constraint additions/retractions.

7.4.3 Experimental results

We present here some first experimental results on dynamic RCPSP. Our experiment consists in comparing our dynamic scheduler with a static scheduler: a scheduling problem P is first optimally solved. Then, an event e is added to this problem. We then compare a re-execution from scratch (as a static scheduler would do) and the dynamic addition of the related constraints in terms of `cpu` time.

We use here some RCPSP test problems[13] introduced by Patterson[14].

For this set of problems, we tried to evaluate the impact of a single modification when comparing static and dynamic scheduling approaches. The following events were evaluated:

- Adding a precedence constraint. Table 1 presents the results obtained on original problems from Patterson from which a single precedence constraint has been removed (which gives the starting problem) then added (which gives the final problem – the original Patterson one[15]).
- Adding a generalized precedence constraint with a fixed delay between two tasks. Table 2 presents the results obtained on original problems from Patterson to which a generalized precedence constraint is added (this constraint is an existing precedence in the optimal solution for which the existing time lag is increased).
- Adding a generalized precedence constraint with a bounded delay between two tasks. Table 3 presents the results obtained on original problems from Patterson to which a generalized precedence constraint is added (this constraint is an existing precedence in the optimal solution for which the existing time lag is increased).
- Adding an overlapping constraint. Table 4 presents the results obtained on original problems from Patterson to which a randomly chosen overlapping constraint is added.

.

In all the tests, activities i and j are randomly selected. In all the tables, we designate by:

- # Act: the number of activities of the problem;
- # Res: the number of resources of the problem;
- t_p: the `cpu` time in seconds needed for solving optimally problem P;
- t_e: the `cpu` time in seconds spent solving this problem after dynamically adding the event e
- t_{pe}: the `cpu` time in seconds needed to obtain an optimal solution of the problem $P \cup \{e\}$ from scratch.
- $\frac{t_{pe}-t_e}{t_{pe}}$: the overall interest of using a dynamic scheduler compared to a static scheduler expressed as the percentage of improvement.

These results clearly show that using a dynamic scheduling solver is of great use compared to solving a series of static problems. In our first experiments, the improvement is never less than 23% and can even reach 98.8 %!

Our second set of reported experiments performed on Kolish, Sprecher and Drexl RCPSP instances[16] considers 4 consecutive modifications (any kind of event[17]) to an original problem. As

13. All experimental data and results are available online at emn.fr/jussien/RCPSP.
14. bwl.uni-kiel.de/Prod/psplib/dataob.html
15. This explains why some results obtained for the static solver are the same: the final problem is the same but not the original one as a different constraint has been removed.
16. wior.uni-karlsruhe.de/RCPSP/ProGen.html
17. See emn.fr/jussien/RCPSP for the details for each instance.

Name	#Act	#Res	t_p	t_e	t_{pe}	$\frac{t_{pe}-t_e}{t_{pe}}\%$
T1P1a	14	3	7.26	1.16	6.96	83.3
T1P1b	22	3	6.75	0.23	6.96	96.7
T1P2	7	3	0.23	0.02	0.19	89.5
T1P3a	13	3	25.82	4.16	12.58	67.0
T1P3b	13	3	15.64	1.21	12.58	90.4
T1P4a	22	3	8.01	1.46	3.63	59.8
T1P4b	22	3	4.50	0.50	3.63	86.2
T1P5a	22	3	20.37	8.70	13.80	**37.0**
T1P5b	22	3	11.34	1.76	13.80	87.3
T1P6a	22	3	135.11	19.60	78.61	75.1
T1P6b	22	3	36.70	7.81	78.61	90.1
T1P6c	22	3	87.80	0.96	78.61	**98.8**
T1P8	9	1	2.11	0.32	2.30	86.1
T1P10	8	2	0.34	0.06	0.25	76.0

Table 1: Adding a precedence constraint

Name	#Act	#Res	t_p	t_e	t_{pe}	$\frac{t_{pe}-t_e}{t_{pe}}\%$
T2P1a	14	3	7.31	0.87	8.84	90.2
T2P1b	14	3	6.58	0.82	9.55	91.4
T2P5a	22	3	14.90	3.45	8.33	58.6
T2P5b	22	3	16.90	3.44	8.19	58.0
T2P5c	22	3	13.82	3.50	8.22	57.4
T2P6a	22	3	62.45	5.14	24.43	79.0
T2P6b	22	3	64.42	1.87	19.56	90.5
T2P6c	22	3	62.50	1.24	23.40	**94.7**
T2P8	9	1	2.19	0.22	2.58	91.5
T2P10	8	2	0.18	0.02	0.26	92.3
T2P11	8	2	0.39	0.20	0.42	**52.4**

Table 2: Adding a generalized precedence constraint

Name	#Act	#Res	t_p	t_e	t_{pe}	$\frac{t_{pe}-t_e}{t_{pe}}\%$
T3P1a	14	3	7.12	0.81	8.50	90.5
T3P1b	14	3	9.69	0.61	9.81	93.8
T3P3	13	3	17.49	0.45	14.34	**96.9**
T3P5a	22	3	14.46	4.82	7.59	36.5
T3P5b	22	3	15.25	3.10	7.81	60.3
T3P6a	22	3	68.38	9.63	19.30	50.1
T3P6b	22	3	75.88	2.52	21.12	88.1
T3P6c	22	3	66.00	17.74	27.34	**35.1**
T3P6d	22	3	65.45	16.25	30.65	47.0
T3P8	9	1	2.90	0.12	2.84	95.8

Table 3: Adding a generalized precedence constraint

Name	#Act	#Res	t_p	t_e	t_{pe}	$\frac{t_{pe}-t_e}{t_{pe}}\%$
T4P1	14	3	7.43	0.38	10.62	**96.4**
T4P5	22	3	15.16	16.39	24.84	34.0
T4P6a	22	3	77.73	8.44	14.60	42.2
T4P6b	22	3	70.45	6.63	18.59	64.3
T4P6c	22	3	64.87	9.85	12.86	**23.4**
T4P8a	9	1	2.37	0.38	2.35	83.9
T4P8b	9	1	2.27	0.12	2.64	95.5

Table 4: Adding an overlapping constraint

12 act./4 res.	Modif. 1	Modif. 2	Modif. 3	Modif. 4
T1KSD1	12.76	26.09	35.84	35.58
T1KSD2	46.24	54.92	62.68	63.13
T1KSD3	35.88	44.12	45.82	45.87
T1KSD5	32.01	67.90	75.15	75.65
T1KSD6	3.81	26.38	46.62	55.92
T1KSD8	22.13	10.64	21.08	22.21
T1KSD9	46.72	52.58	47.75	49.27
T1KSD10	-29.21	-0.35	15.74	30.27

Table 5: Some Kolish, Sprecher and Drexl instances (4 consecutive dynamic events). Relative speed-up (in %)

22 act./4 res.	Modif. 1	Modif. 2	Modif. 3	Modif. 4	Modif. 5
T2KSD11	47.15	12.49	5.87	6.89	-1.98
T2KSD13a	34.81	43.55	43.04	49.89	50.98
T2KSD13b	39.19	-126.79	-174.22	-216.67	-146.42
T2KSD15	-47.62	-42.50	-41.00	-65.06	-88.20
T2KSD17a	-2.40	1.65	-5.90	-16.23	-43.06
T2KSD17b	3.58	13.11	-12.76	-10.51	-74.91
T2KSD18a	30.61	26.76	-52.76	-19.07	0.85
T2KSD18b	40.39	56.09	40.54	42.56	45.15

Table 6: Other Kolish, Sprecher and Drexl instances (5 consecutive dynamic events). Relative speed-up (in %)

we can see in Table 5, our results are quite promising. Even bad results (instance `T1KSD10`) get better in the long run. However, notice that some results (see Table 6 – results for 5 consecutive modifications) show that dynamic handling is not always the panacea and rescheduling from scratch can be very quick.

8. Explanations for search in constraint programming

Looking at the way variables and constraints interact together through explanations [36] leads to consider a more intricate usage of explanations within constraint programming languages. Therefore, one can design new programming gimmicks and new approaches. These new approaches are designed under the name explanation-based constraint programming (*e-constraints* for short).

8.1 From backtrack-based to explanation-based solving

Most CSP solving algorithms are derived from a backtrack-based complete search. The drawbacks of this approach have been known for a long time: thrashing and backtracking to irrelevant choice points. The previously developed fancy backtrackers were not really convincing to address this issue (time or space overhead, no real advantages) [12]. Nevertheless, more recently, the `mac-dbt` algorithm [51] showed that explanation-based algorithms could compete well with backtrack-based algorithms in real-world situations. The `mac-dbt` algorithm is to `dbt` what `mac` is to `sb`. It can be described using a very generic algorithm that appears as the archetype of explanation-based constraint solving.

8.1.1 From standard backtracking to dynamic backtracking

Most complete search algorithms over Constraint Satisfaction Problems (CSP) are based on *Standard Backtracking* (`sb`): a depth-first search is performed using chronological backtracking. Various *intelligent* backtrackers have been proposed: *Conflict-directed BackJumping* (`cbj`) [67], *Dynamic Backtracking* (`dbt`) [37], *Partial order Dynamic Backtracking* (`pdb`) [38], *Generalized Dynamic Backtracking* (`gpb`) [14], etc. In these algorithms, information (namely a special case of explanations: nogoods) is kept when encountering inconsistencies so that the forthcoming search will not get back to already known traps in the search space.

Dependency Directed Backtracking (`ddb`) [76] was the first algorithm to use this enhancement, however it has an important drawback: its space complexity is exponential since the number of nogoods it stores increases monotically. To address this problem, algorithms such as `cbj` or `dbt` eliminate nogoods that are no longer relevant to the current variable assignment. By doing so, the space complexity remains polynomial.

When a failure occurs, these algorithms have to identify the assignment to be reconsidered (suspected to be a **culprit** for the failure).

- `sb` always considers the most recent assignment to be a culprit. This selection may be completely irrelevant for the current failure leading to useless exploration of parts of the search tree already known to be dead-ends (thrashing).

- In `cbj` a conflict-set is associated to each variable: CS_{v_i} (for the variable v_i) contains the set of the assigned variables whose value is in conflict with the value of v_i. When identifying a dead-end while assigning v_i, `cbj` considers the most recent variable in CS_{v_i} to be a culprit. A backtrack then occurs: the conflict-sets and domains of the *future* variables are reset to their original value. By doing so, `cbj` forgets a lot of information that could have been useful. This also leads to thrashing.

- `dbt` selects the most recent variable in the *computed* nogood (the conflict-set of `cbj`) in order to undo the assignment. However, thanks to the explanations, `dbt` only removes related information that depends on it and so avoids thrashing: useful information is kept. Indeed, there is no real backtracking in `dbt` and, like in a repair method, only the assignments that caused the contradiction are undone.

Notice that `sb` can also be considered as selecting the most recent assignment of a nogood, namely the nogood that contains all the current variable assignments (which fails to give really relevant information).

8.1.2 Integrating constraint propagation within dynamic backtracking

Constraint propagation has been included in `sb` leading to forward checking `fc` and to the *Maintaining Arc-Consistency* algorithm (`mac`) [71]. `mac` is nowadays considered as one of the best algorithms for solving CSP [12].

Several attempts to integrate constraint propagation within intelligent backtrackers have been made: for example, Prosser has proposed `mac-cbj` which maintains arc consistency in `cbj` [67]. But, Bessière and Régin [12] have stopped further research in that field by showing that `mac-cbj` was very rarely better than `mac`. They concluded that there was no need to spend time nor space on intelligent backtracking because the brute force of `mac` simply does it more quickly. From our point of view, the inadequacy of `mac-cbj` is more related to the fact that `cbj` does not avoid thrashing[18] than to the cost of the management of nogoods. When backtracking occurs, `cbj` comes back to a relevant assignment, and then forgets all the search space developed since this assignment has been performed: like `sb`, `cbj` has a **multiplicative** behavior on independent sub-problems. `dbt` does not

18. A thrashing behavior consists in repeatedly performing the same search work due to the backtrack mechanism.

```
function solve(pb: Problem): boolean
(1)  begin
(2)    unassignedVars ← pb.vars
(3)    try (
(4)       while not(empty(unassignedVars))
(5)           let idx ← nextVarToAssign(pb)    // variable choice
(6)               v ← unassignedVars[idx]
(7)               a ← selectValToAssign(pb, v)    // value choice
(8)           in (
(9)               try (
(10)                   unassignedVars :delete v
(11)                   post(pb, v == a)     // instantiation
(12)                   propagate(pb)
(13)               )
(14)               catch LabelingContradiction     // An empty domain found
(15)                   handleContradiction(pb)     // classically: BACKTRACK
(16)               )
(17)          )
(18)      endwhile
(19)      true
(20)   )
(21)   catch ProblemContradiction
(22)      false    // No solution
(23)   )
(24) end
```

Figure 4: Solving a CSP

only use nogoods to perform *intelligent* backtracking but also to avoid thrashing and so becomes **additive** on independent sub-problems [37]. [12] had another point preventing the use of nogoods: it is always possible to find an intelligent labeling heuristic so that a *standard backtracking*-based algorithm will perform a search as efficiently as an intelligent backtracker. In our experience, using a good heuristic reduces the number of problems on which the algorithm thrashes but does not make it additive on independent subproblems: there are still problems on which the heuristic cannot prevent thrashing.

In [51], we introduced the `mac-dbt` algorithm which shows the efficient integration of constraint propagation within `dbt` thanks to the use of explanations. The general behavior of `mac-dbt` is described in the algorithm in Figure 4: as long as no solution has been found, choose a variable and a value for it, assign the value to the variable (line 11), propagate this new information (line 12). If a contradiction occurs during that process, use a special handling contradiction procedure (line 15 and the algorithm in Figure 5).

This special handling contradiction procedure merely amounts to a backtrack if a standard backtrack-based search is desired. However, if one wants to benefit from an explanation-based solver, an explanation for this failure can be computed (see line 2 in the algorithm in Figure 5) and, if possible, a choice (an assignment constraint) to undo[19] is selected (line 7). At this point, an intelligent backtracker can be defined by performing a classical backtrack to that selected choice. In order to use explanations as much as possible, a dynamic constraint removal should be performed (line 12), followed by the posting within a given context[20] of the negation of the undone decision and the propagation of the new information (line 15). It is important to recursively handle any contradiction that may appear (line 18).

19. Notice that this choice is tightly guided by the completeness requirements of the algorithms (see [14]).
20. Line 14 in the algorithm in Figure 5 introduces the constraint opposite(ct) which will remain active as long as the context e remains valid. See [51] for more information on that point.

```
procedure handleContradiction(pb:  Problem)
(1)  begin
(2)    let e ← becauseOf(theDom(getFailingVariable(pb)))     // conflict explanation
(3)    in (
(4)       if e empty then
(5)          raiseProblemContradiction()
(6)       else
(7)          let ct ← selectConstraint(e)     // select a to be removed choice
(8)          in (
(9)               if ct exists then
(10)                   unassignedVars :add ct.v1
(11)                   try (
(12)                        remove(ct)     // perform constraint removal
(13)                        e :delete ct
(14)                        post(pb, opposite(ct), e)     // the context-guarded negation is added
(15)                        propagate(pb))     // achieving consistency
(16)                   )
(17)                   catch LabelingContradiction
(18)                        handleContradiction(pb)     // recursive handling
(19)                   )
(20)              else
(21)                   raiseProblemContradiction()
(22)              endif
(23)         )
(24)      endif
(25)   )
(26) end
```

Figure 5: Contradiction handling

8.1.3 A generic algorithm

One can identify three different components[21] in the algorithms in Figures 4 and 5 that help understand and generalize their behavior:

- a **propagation** component that is used to propagate information throughout the constraint network when a decision is made during search. Two operators are needed: `filter`ing and `check`ing if a solution can exist.
- a **learning** component that is used to make sure that the search mechanism will avoid (as much as possible) getting back to states that have been explored and proved to be solutionless. Using a rough analogy with the brain, we will use two operators: a `record`ing operator that learns new pieces of information and a `forget`ting operator that will make room for new information to be learnt.
- a **moving** component whose aim is, unlike the other two components, to explore the search space instead of pruning it. There are two moving operators: `repair` to be used when the current constraint system is contradictory and needs some modification and `extend` to potentially add new information when no contradiction has yet been detected but when no solution has been found.

These three components can be used to design a generic CSP solving algorithm (the PLM algorithm – see Figure 6) that encompasses complete and incomplete searches, prospective and retrospective algorithms [54, 53]:

- search starts from an initial set of decision constraints that may range from the empty set (typically for backtrack-based search) or a total assignment (typically for a local search algorithm)

21. A formal description of these components can be found in [53].

```
procedure PLM(V,C,C_D)
(1)  begin
(2)     P ← {V, C, C_D}
(3)     repeat
(4)        P ← filter(P)
(5)        switch check(P)
(6)           case no solution :
(7)              P ← forget(repair(record(P)))
(8)           case solution found :
(9)              return P
(10)          case not enough information :
(11)             P ← extend(P)
(12)       endswitch
(13)    until conditions of termination
(14) end
```

Figure 6: A generic algorithm for the PLM components

- decisions are made (`extend`) and propagated (`filter`) until a contradiction occurs
- when a contradiction does occur (line 6), the information related to the dead-end (*e.g.* a conflict explanation) is learnt (`record`), the current state is repaired (`repair`) and some information is *forgotten* (`forget`).
- search terminates as soon as a solution is found (line 8) or the conditions of termination (line 13) are fulfilled. Conditions of termination can be for example a maximum number of iterations, the exhibition of a proof that no solution exists, etc.

Using explanations greatly helps to implement concretely such a generic framework. Indeed, explanations can be used to determine precise conflict explanations, perform dynamic reparations of any constraint system, etc. The PALM system effortlessly implements this generic framework.

8.2 A new family of algorithms: decision-repair

The PLM generic algorithm has been used to describe several well-known algorithms [53]:

- systematic algorithms such as `sb`, `cbj`, `dbt`, `mac`, `mac-dbt`, etc.
- non-systematic algorithms such as `tabu search` [39], `gsat` [74], etc.

The PLM generic algorithm has also been used to design new algorithms: the `decision-repair` family [52]. The idea of `decision-repair` is to combine the propagation-based nature of `mac-dbt` and the true freedom (in the search space exploration) given by a local search algorithm such as `tabu search`. Therefore, in terms of the PLM generic algorithm, we have:

- the starting set of decision constraints is empty;
- `filter` uses a standard filtering algorithm for reducing the domain of the variables of the problem;
- `record` computes an explanation-set for the current contradiction and stores it in a *tabu* list of fixed size K;
- `forget` erases the oldest stored explanation-set if the *tabu* list is full;
- `extend` classically adds new decisions (variable assignment, domain splitting, etc.) as long as no solution has been found yet;

- `repair` heuristically selects a decision to undo from the last computed explanation-sets (and whose negation is compatible with the stored explanation-sets).

As several parameters remain unspecified (the way of handling the *tabu* list, the heuristic to be used to select decisions to undo, etc.), `decision-repair` is a family of algorithms. However, the two main points of this algorithm are: it makes use of a repair algorithm (local search) as a basis, and it works on a partial instantiation in order to be able to use filtering techniques.

A comprehensive study of the behavior of `decision-repair` has shown that the key components of this algorithm are: its explanation-directed heuristics and its ability both to perform a local search and to prune the search space [52]. Experiments with `decision-repair` have shown good results for open-shop scheduling problems (see Section 9.3).

9. A case study: solving open-shop scheduling problems

We have experimented using explanation-based constraint programming algorithms for solving open-shop scheduling problems for several years now. First we developed intelligent backtrackers that provided good first results and greatly improved our techniques using more explanation-based features of e-constraints. This section provides some details about these results.

9.1 Open-shop scheduling problems

Classical scheduling shop problems, for which a set J of n jobs consisting each of m tasks (operations) must be scheduled on a set M of m machines, can be considered as CSP[22]. One of these problems is called the open-shop problem [40]. For this problem, operations for a given job may be sequenced as wanted but only one at a time. We will consider here the building of non-preemptive schedules of minimal makespan[23].

The open-shop scheduling problem is NP-hard as soon as $\min(n, m) \geq 3$. This problem although quite simple to enunciate is really hard to solve optimally: instances of size 6×6 (*i.e.* 36) variables remain unsolved !

Only two branch-and-bound methods for this problem have been published so far. The first one [16] is based on the resolution of a one-machine problem with positive and negative time-lags. The second one, [18], consists, in each node, in fixing disjunctions on the critical path of a heuristic solution. It combines two concepts:

- a generalization of a branching scheme first introduced by Grabowski *et al.* [41] for one-machine problems with release dates and due dates;
- *immediate selections* [19], a method initially designed to fix disjunctions in Job-Shop problems.

We used this method as a basis for our experiments and enhanced it in two steps:

1. designing an intelligent backtracker by using an explanation-based constraint solver to propagate new decisions made during search;
2. using the `decision-repair` family of algorithms to design a new efficient heuristic for solving open-shop scheduling problems.

9.2 An intelligent backtracker

In [43], we presented an *intelligent backtracker* for solving classical *open-shop* scheduling problems. This first use of an explanation-based search led us to solve an open instance for the first time.

22. The variables of the CSP are the starting dates of the tasks. Bounds thus represent the least feasible starting time and the least feasible ending time of the associated task.
23. Ending time of the last task.

Table 7 presents the results obtained on Taillard's benchmarks [77]. These benchmarks are composed of 10 square instances of size 4, 5, 7 and 10. Among the problems of size 10, 4 were still unsolved at that time[24].

For each problem, a lower bound LB, an upper bound UB (initial solution) and the optimal solution OPT (unknown for four problems of size 10) are given. This table also contains the number of backtracks. We stopped the search at 250 000 backtracks. If the optimal solution is not reached at that limit, the best solution found is indicated between brackets. The optimal solution of the open problem we solved is outlined.

Although it is difficult to generalize, the results having been obtained on only 40 problems, it seems that the bigger the problem, the more efficient the technique is: indeed, this technique has no effect on the 4x4 problems of Taillard, the number of backtracks is reduced for 9 problems of size 5, and for all problems of size 7x7.

For 10x10 problems, the improvement is very large: three problems are solved by our technique, among which is an open problem (the one with outlined optimal value) in only 4843 backtracks, whereas without intelligent backtracking, no problem of that size is solved in less than 250 000 backtracks. And for the other 10x10 problems, our technique generally provides better solutions at the limit of backtrack numbers.

As regards execution times, the version with intelligent backtracking is about two times slower than the initial version in each node. But this waste of time is widely compensated by the decreased number of explored nodes. Thus for the open problem we solved, the execution time is reduced by at least 90% (since the problem is not solved after 250 000 backtracks in the initial version).

These very good first results led us to go further in using explanations and to fully use the power of e-constraint programming.

9.3 An efficient heuristic technique

Because the open-shop problem is very hard to solve exactly, various heuristics have been proposed[25]: greedy heuristics such as specific list heuristics [44] or local searches such as highly specialized tabu searches [4, 61] or genetic algorithms [64], etc.

We tried `decision-repair` on the open-shop problem using one of its implementations described below:

Filtering technique Precedence constraints are handled with 2B-consistency filtering [23, 60] and resource usage constraints are handled through *task-intervals* [20] (see also Section 4.2.2).

Search strategy For shop scheduling problems, enumeration is usually performed on the relative order in which tasks are scheduled on the resources. The decision constraints are thus precedence constraints between tasks[26]. We use the branching scheme introduced in [18] to select such decisions.

Tabu list The implementation uses a tabu list of size 7.

Repair The `repair` function we used records in the tabu list the newly computed contradiction explanation k (it is a conflict). It tries to find one decision in k such that negating this decision makes the decision set compatible with all the stored conflicts. When several decisions can be negated, we use the following weighting-conflict heuristics: a weight is associated with each decision; the weight characterizes the number of times that the decision has appeared in any conflict. The `repair` function chooses to negate the decision with the greatest weight that,

24. This is no longer true
25. We mention here heuristics that are considered to be the best techniques to solve open-shop scheduling problems.
26. When every possible precedence has been posted, setting the starting date of the variable to its smallest value provides a feasible solution.

Size	OPT	LB	UB	without intelligent backtracking number of backtracks	with intelligent backtracking number of backtracks
4	193	186	197	18	18
4	236	229	253	37	37
4	271	262	272	29	29
4	250	245	260	26	26
4	295	287	305	55	55
4	189	185	193	19	19
4	201	197	203	22	22
4	217	212	217	14	14
4	261	258	268	32	32
4	217	213	224	31	31
5	300	295	303	273	**270**
5	262	255	266	252	**238**
5	323	321	347	943	**940**
5	310	306	319	678	678
5	326	321	330	840	**836**
5	312	307	325	756	**737**
5	303	298	308	420	**411**
5	300	292	304	853	**851**
5	353	349	368	1000	**987**
5	326	321	337	2377	**2377**
7	435	435	452	2840	**1860**
7	443	443	465	18196	**16862**
7	468	468	495	98903	**96502**
7	463	463	482	1494	**1410**
7	416	416	425	317	**256**
7	451	451	471	21492	**20364**
7	422	422	449	56944	**53773**
7	424	424	433	1939	**1552**
7	458	458	476	560	**535**
7	398	398	411	731	**710**
10		637	654	$>$ 250000 (644)	$>$ 250000 (644)
10	588	588	600	$>$ 250000 (594)	$>$ 250000 **(591)**
10		598	611	$>$ 250000 (610)	$>$ 250000 **(604)**
10	577	577	588	$>$ 250000 (584)	**26777**
10	640	640	661	$>$ 250000 (658)	$>$ 250000 (658)
10	538	538	544	$>$ 250000 (544)	$>$ 250000 (544)
10	**616**	616	633	$>$ 250000 (633)	**4843**
10	595	595	604	$>$ 250000 (604)	$>$ 250000 (604)
10	595	595	612	$>$ 250000 (597)	**245100**
10		596	612	$>$ 250000 (611)	$>$ 250000 **(606)**

Table 7: Results of our intelligent backtracker on Taillard's problems. **Bold** figures indicate improvements

```
procedure minimise-makespan(C)
(1)  begin
(2)     C_D ← initial decision set
(3)     bound ← +∞
(4)     lastSolution ← failure
(5)     repeat
(6)        C ← C ∪ { makespan < bound}
(7)        solution ← decision-repair(C)
(8)        if solution = failure then
(9)           return lastSolution
(10)       else
(11)          bound ← value of makespan in solution
(12)          lastSolution ← solution
(13)       endif
(14)    until false
(15) end
```

Figure 7: Algorithm used to solve open-shop problems

when negated, makes the new decision set compatible with all the conflicts in the tabu list. If such a decision does not exist, it is considered as a stopping criterion for the overall algorithm.

Stopping criterion The failure conditions specifying the exit of `decision-repair` are either a *stop* returned by the `repair` function or 3000 iterations without improvement since the last solution reached.

Minimization of the makespan The open-shop problems we consider are optimization problems. This requires a main loop that calls `decision-repair` until improvement is no longer possible. (See Figure 7.) Improvements are forced by adding a constraint that specifies that the makespan is less than the current best solution found. The initial decision set for each call of the function `decision-repair` is the latest set of decisions (which defines the last solution found).

We studied three series of reference problems:

1. **Taillard**'s problems [77] (see section 9.2).
2. **Brucker *et al.*** problems [17]: 52 problems of size 3×3 to 8×8. These problems are characterized by a common LB (the classical lower bound[27]) value: 1000.
3. **Guéret and Prins**' problems[28]: Those 80 problems (8 series of 10 problems of size 3×3 to 10×10) have been generated using results presented in [45] for generating really hard open-shop instances. They all share a common LB (classical lower bound) value and the fact that another lower bound [45] gives a much greater value.

`decision-repair` (referred to as **TDR** in the results) is compared with the best published solving techniques for the open-shop problem:

- For Taillard's instances, our results are compared with two highly specialized tabu searches tailored for solving open-shop problems, one presented in [4] (referred to as **TS-A97** in the results) and one presented in [61] (referred to as **TS-L98** in the results).
- For all the instances, our results are also compared with a genetic algorithm introduced in [64] (referred to as **GA-P99** in the results) which gives very good results on all these problems.

27. Maximum load of the involved machines and jobs.
28. Available at `emn.fr/gueret/OpenShop/OpenShop.html`.

Figure 8 presents the results obtained on Taillard's problems, Figure 9 on Brucker's instances, and finally Figure 10 on Guéret and Prins' problems. Cpu time is not available in [61], [4], nor in [64][29]. *Average* cpu time for `decision-repair` is not really significant since cpu time strongly depends on the instance of the problem. Just to give an idea, for Taillard's instances, 10×10 average cpu time is 15 hours and for 7×7 average cpu time is 2 hours. For Brucker's instances and Guéret and Prins' problems, 10×10 average cpu time is 3 to 4 hours and, for size less than 8×8, average cpu time is less than 4 minutes.

Series	TS-L98	TS-A97	GA-P99	TDR
4×4	0 / 0 (10)	(*)	0.31 / 1.84 (8)	0 / 0 (10)
5×5	0.09 / 0.93 (9)	(*)	1.26 / 3.72 (1)	0 / 0 (10)
7×7	0.56 / 1.77 (6)	0.75 / 1.71 (2)	0.41 / 0.95 (4)	0.44 / 1.92 (6)
10×10	0.29 / 1.41 (6)	0.73 / 1.67 (1)	0 / 0 (10)	2.02 / 3.19 (0)

Table 8: **Results on Taillard's instances.** Results are presented in the following format: "average deviation from the optimal value" / "maximum deviation from the optimal value" ("number of optimally solved instances") **(*)** Only results for 7 problems of size 7×7 and 3 of size 10×10 are given in the paper.

Series	GA-P99	TDR
3×3 (8 pbs)	0 / 0 (8)	0 / 0 (8)
4×4 (9 pbs)	0 / 0 (9)	0 / 0 (9)
5×5 (9 pbs)	0.36 / 2.07 (6)	0 / 0 (9)
6×6 (9 pbs)	0.92 / 2.27 (3)	0.71 / 3.50 (6)
7×7 (9 pbs)	3.82 / 8.20 (6)	4.40 / 11.5 (5)
8×8 (8 pbs)	3.10 / 7.50 (8)	4.95 / 11.8 (2)

Table 9: **Results on Brucker's instances.** Results are presented according to the following format: "average deviation from the optimal value" / "maximum deviation from the optimal value" ("number of optimally solved instances") except for 7×7 and 8×8 time for which the deviation is computed from the LB value (1000 for each problem).

Series	BB-G00	GA-P99	TDR	Open instances	TDR yield
3×3	10 / 10	10 / 10	10 / 10	0	-
4×4	10 / 10	10 / 10	10 / 10	0	-
5×5	10 / 10	8 / 8	10 / 10	0	-
6×6	9 / 7	2 / 1	10 / 8	3	1 / 1
7×7	3 / 1	6 / 3	10 / 4	9	1 / 3
8×8	2 / 1	2 / 1	10 / 4	9	3 / 7
9×9	1 / 1	0 / 0	10 / 2	9	1 / 9
10×10	0 / 0	5 / 0	5 / 0	10	0 / 5

Table 10: **Results on Guéret and Prins' problems.** Results are presented according to the following format: "number of problems solved giving the best results" / "number of optimally solved problems". **BB-G00** reports the results of our intelligent backtracker described in Section 9.2 and stopped after 350 000 backtracks (which represents around 24 hours of cpu time). What `tabu decision-repair` gave to the solving of these problems (TDR yield) is indicated by "the number of closed instances" / "the number of newly improved instances"

29. This is quite usual for open-shop scheduling results. Indeed, as the problem itself is really hard, what is important is the quality of the solution and not the time required to obtain it. Moreover, in real-life applications such as satellite scheduling problems [65] improving a solution by one unit can save so much money that satellite operators are ready to wait as long as a full day for that improvement!

Recall that `decision-repair` is a generic algorithm that has been instantiated simply to solve a very specific problem which has its own research community. The results obtained on the three sets of problems are therefore very interesting because they show that our algorithm is a competitive algorithm compared with the other techniques.

As far as Taillard's instances are concerned, `decision-repair` gives comparable results but the *tabu search* of [61] is still the best technique except for 10×10 problems where the genetic algorithm shows the best results.

On Brucker's instances, `decision-repair` is far better than the genetic algorithm on small instances but the latter becomes better on larger problems.

For the third set of problems (the really hard instances of Guéret and Prins) `decision-repair` shows all the interest of combining local search and constraint propagation: `decision-repair` closed[30] 6 of these instances. Furthermore, it provided new best results for 19 other instances; thus it improved known results for 25 instances out of 40 open ones.

Up to size 9×9, `decision-repair` gives far better results than both the genetic algorithm and branch-and-bound search (that has been truncated by a time criterion). For 10×10 problems, `decision-repair` is still better than the branch-and-bound but is matched by the genetic algorithm.

Such a good behavior of `decision-repair` was quite surprising because, unlike the other specialized algorithms, our implementation remains general and does not need any tuning of complex parameters. This is probably due to the search used for the open-shop problem, which dynamically builds independent sub-problems by adding precedence constraints: classical backtracking algorithms may start by partially solving a sub-problem, then go to another one, solve it, and then continue to solve the first sub-problem. In cases where it has to backtrack to choices in the first part of its work, the search spaces of the two sub-problems are multiplied. `decision-repair`, thanks to its use of explanations, can identify independent sub-problems and stay in a sub-problem until it has been solved. Also the heuristic we have introduced seems to be good. Once again, this is another benefit from the use of explanations.

9.4 Analysis

We performed a comprehensive study of the behavior of `decision-repair` in [52]. It showed that the key components of this algorithm are its conflict-directed heuristics and its ability both to perform a local search and to prune the search space. These results clearly advocate the use of explanations within constraint programming for providing new techniques and new programmation gimmicks.

Summary

In this fourth part, we introduced possible usage of explanations within constraint programming in a gradated way: from immediate usage to in-depth usage. First, we showed how to use explanation to provide user information and addressed the user-accessibility issue of produced explanations. Second, we introduced the use of explanations for solving dynamic problems: incremental correct constraint retraction, implementation issues, and over-constrained problems handling. Third, we introduced a more in-depth use of explanations for improving search in constraint programming whose main result is the e-constraints *philosophy: a new way of considering search within constraint programming (decomposing search into three components: propagation, learning, and move). Moreover, we introduce the* `decision-repair` *family of algorithms which provides an original way of combining constraint propagation and local search. Finally, we illustrated the application of our work to the solving of open-shop scheduling problems.*

30. An optimal solution was found and proved – a lower bound is known – for the first time.

v. Conclusion and further work

In this document, we have presented an introduction to explanations in constraint programming and shown how to compute and use them. We introduced a formal framework for constraint programming in which we defined our notion of explanation. We detailed different levels and different quality of explanations and related them to previous concepts such as justifications and conflict-sets. Moreover, we introduced different possible ways of computing explanations from off-line techniques to on-line techniques. We described both an intrusive approach (the one provided by the PALM system) and a promising non-intrusive technique based on aspect-oriented programming.

We introduced possible usage of explanation within constraint programming. We first promote explanations as a tool for the user information during/after solving a constraint problem (pointing out some open issues on user interaction) and then introduced the use of explanations for handling dynamic constraint solving (from user interaction to automatic handling of over-constrained problems) through the example of solving dynamic scheduling problems.

We have also illustrated how programming habits are changed when a complete use of explanations is made leading to a general framework for designing new search algorithms: the PLM approach which decomposes search techniques into three components (propagation, learning, and move). Such an explanation-based constraint programming approach led us to design two new efficient algorithms: a complete one (`mac-dbt`) and its incomplete generalization (the `decision-repair` family) which harmoniously mixes local search and constraint propagation. Finally, we illustrated the use of our new techniques for solving open-shop scheduling problems.

This recent field in constraint programming opens up several perspectives:

- regarding **explanation computation**, the following topics spring to mind: finding mechanisms or enumeration orderings that are efficient for producing both solutions and explanations; computing explanations in a lazy way (keeping less information – as in [11, 25]); designing constraint propagation algorithms with explanations in mind and therefore being able to provide a formal presentation of the explanations together with the operational semantics of the constraints; investing in monitoring explanation computation systems (like our proposition with AOP in Section 5.2.1) which can really help the diffusion of that technology; or *investing* in explanations and radically modifying solving mechanisms;

- regarding **user interaction**, our user-friendly explanations are only a first step towards making explanations accessible to anyone: the tree-based problem representation of our user-friendly explanations is a limitation; much work remains to be done: for example, one could consider several ontologies to be used for clustering the information contained in an explanation in concepts that the current user would understand.

- regarding **explanation-based constraint programming**, `mac-dbt` and `decision-repair` are efficient explanation-based search algorithms. The potential of explanations has not been completely explored yet (especially within the PLM framework) and should provide new powerful algorithms.

We strongly believe that explanations for constraint programming is a really a hot topic for the future. Notice that there now exists a web portal on the subject: `e-constraints.net`.

References

[1] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[2] M. Ågren. Tracing and explaining the execution of clp(fd) programs in SICStus prolog. Master's thesis, Computer Science Department, Uppsala University, Sweden, July 2002.

[3] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *16th IEEE conference on Automated Software Engineering (ASE'01)*, pages 166–173, San Diego, USA, November 2001. IEEE Computer Society Press.

[4] D. Alcaide, J. Sicilia, and D. Vigo. A tabu search algorithm for the open shop problem. *TOP : Trabajos de Investigación Operativa*, 5(2):283–296, 1997.

[5] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic csps - application to configuration. *Artificial Intelligence*, 135(2002):199–234, 2002.

[6] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.

[7] R. Bakker, F. Dikker, F. Tempelman, and P. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proceedings IJCAI'93*, pages 276–281, 1993.

[8] R. J. Bayardo Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[9] F. Benhamou. Heterogeneous constraint solving. In M. Hanus and M. Rofríguez-Artalejo, editors, *International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 1996.

[10] P. Berlandier and B. Neveu. Arc-consistency for dynamic constraint problems: A rms-free approach. In T. Schiex and C. Bessiére, editors, *Proceedings ECAI'94 Workshop on Constraint Satisfaction Issues raised by Practical Applications*, Amsterdam, August 1994.

[11] C. Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[12] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In *CP'96*, Cambridge, MA, 1996.

[13] J. Blazewicz, J. Lenstra, and A. R. Kan. Scheduling projects subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[14] C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.

[15] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In G. Levi and M. Martelli, editors, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, June 1989. MIT Press.

[16] P. Brucker, T. Hilbig, and J. Hurink. A branch and bound algorithm for scheduling problems with positive and negative time-lags. Technical report, Osnabrueck University, May 1996.

[17] P. Brucker, B. Jurish, B. Sievers, and B. Wöstmann. A branch and bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76:43–49, 1997.

[18] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107:272–288, 1998.

[19] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[20] Y. Caseau and F. Laburthe. Improving clp scheduling with task intervals. In P. V. Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming, ICLP'94*, pages 369–383. MIT Press, 1994.

[21] Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[22] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions mathematical foundation. In *Symposium on Artificial Intelligence and Programming Languages*, volume 12(8) of *ACM SIGPLAN Not.*, pages 1–12, 1977.

[23] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(2):281–331, 1987.

[24] N. de Siqueira and J. Puget. Explanation-based generalisation of failures. In *Proceedings ECAI'88*, pages 339–344, Munich, Germany, 1988.

[25] R. Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In 8^{th} *Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, Toulouse, France, 1996.

[26] R. Debruyne, G. Ferrand, N. Jussien, W. Lesaint, S. Ouis, and A. Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03: Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 172–176, St. Augustine, Florida, USA, May 2003. AAAI press.

[27] R. Douence and N. Jussien. Non-intrusive constraint solver enhancements. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS'02)*, volume 344 of *CW Reports*, pages 26–36, July 2002.

[28] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In *Reflection*, pages 170–186, 2001.

[29] A. Elkhyari, C. Guéret, and N. Jussien. Conflict-based repair techniques for solving dynamic scheduling problems. In *Principles and Practice of Constraint Programming (CP 2002)*, number 2470 in Lecture Notes in Computer Science, pages 702–707, Ithaca, NY, USA, September 2002. Springer-Verlag. Short paper.

[30] F. Fages, J. Fowler, and T. Sola. A reactive constraint logic programming scheme. In *International Conference of Logic Programming, ICLP'95*, Tokyo, 1995.

[31] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.

[32] G. Ferrand, W. Lesaint, and A. Tessier. Theoretical foundations of value withdrawal explanations for domain reduction. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[33] E. C. Freuder, C. Likitvivatanavong, and R. J. Wallace. A case study in explanation and implication. In *CP2000 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, 2000.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[35] Y. Georget, P. Codognet, and F. Rossi. Constraint retraction in clp(fd): Formal framework and performance results. *Constraints, an International Journal*, 4(1):5–42, 1999.

[36] M. Ghoniem, N. Jussien, and J.-D. Fekete. Visualizing explanations to exhibit dynamic structure in constraint problems. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, Kinsale, Ireland, September 2003.

[37] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[38] M. Ginsberg and D. A. McAllester. Gsat and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation (KR94)*, pages 226–237, 1994.

[39] F. Glover and M. Laguna. *Modern Heuristic Techniques for Combinatorial Problems, chapter Tabu Search, C. Reeves*. Blackwell Scientific Publishing, 1993.

[40] T. Gonzales and S. Sahni. Open-shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, 1976.

[41] J. Grabowski, E. Nowicki, and S. Zdrzalka. A block approach for single-machine scheduling with release dates and due dates. *European Journal of Operations Research*, 26:278–285, 1986.

[42] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, pages 57–64, Seattle, WA, USA, August 2001.

[43] C. Guéret, N. Jussien, and C. Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, 2000.

[44] C. Guéret and C. Prins. Classical and new heuristics for the open-shop problem. *European Journal of Operations Research*, 107(2):306–314, 1998.

[45] C. Guéret and C. Prins. A new lower bound for the open-shop problem. *AOR (Annals of Operations Research)*, 92:165–183, 1999.

[46] U. Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.

[47] N. Jussien. e-constraints: explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 2001.

[48] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[49] N. Jussien and P. Boizumault. Implementing constraint relaxation over finite domains using ATMS. In M. Jampel, E. Freuder, and M. Maher, editors, *Over-Constrained Systems*, number 1106 in Lecture Notes in Computer Science, pages 265–280. Springer-Verlag, 1996.

[50] N. Jussien and P. Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.

[51] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[52] N. Jussien and O. Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, July 2002.

[53] N. Jussien and O. Lhomme. Unifying search algorithms for CSP. Research Report 02-3-INFO, École des Mines de Nantes, Nantes, France, 2002.

[54] N. Jussien and O. Lhomme. Combining constraint programming and local search to design new powerful heuristics. In *5th Metaheuristics International Conference (MIC'2003)*, Kyoto, Japan, August 2003.

[55] N. Jussien and S. Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.

[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[57] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, 1997. Springer Verlag.

[58] R. Klein and A. Scholl. Computing lower bounds by destructive improvement: an application to Resource-Constrained Project Scheduling Problem. *European Journal of Operational Research*, 112:322–345, 1999.

[59] F. Laburthe. CHOCO: implementing a CP kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.

[60] O. Lhomme. Consistency techniques for numeric CSPs. In *IJCAI'93*, pages 232–238, Chambéry, France, August 1993.

[61] C.-F. Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26, 1998.

[62] S. Ouis, N. Jussien, and P. Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03: Sixteenth international Florida Artificial Intelligence Research Society conference*, pages 192–196, St. Augustine, Florida, USA, May 2003. AAAI press.

[63] G. Pesant. A filtering algorithm for the stretch constraint. In Springer-Verlag, editor, *Principles and Practice of Constraints Programming (CP 2001)*, number 2239 in Lecture Notes in Computer Science, pages 183–195, Paphos, Cyprus, 2001.

[64] C. Prins. Competitive genetic algorithms for the open shop scheduling problem. *Mathematical Methods of Operations Research*, 52(3):389–411, 2000.

[65] C. Prins and J. Carlier. Resource optimization in a TDMA/DSI system: the eutelsat approach. In *Proceedings of the International Conference on Digital Satellite Communications (ICDSC 7)*, pages 511–518, Munich, Germany, 1986.

[66] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Strathclyde, 1991).

[67] P. Prosser. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde, 1995.

[68] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.

[69] G. Rochart and N. Jussien. Explanations for global constraints: instrumenting the stretch constraint. Research Report 03-01-INFO, École des Mines de Nantes, Nantes, France, 2003.

[70] G. Rochart, N. Jussien, and F. Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, Kinsale, Ireland, September 2003.

[71] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

[72] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[73] T. Schiex and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In C. Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[74] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92: Proceedings 10th National Conference on AI*, pages 440–446, San Jose, July 1992.

[75] M. H. Sqalli and E. C. Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI: National Conference on Artificial Intelligence*, pages 318–325, 1996.

[76] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[77] É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[78] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[79] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[80] M. Wilson and A. Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16(3):277–318, July 1993.